Multiple-Precision Binary Comparison (MPBCMP)

6J

Compares two multi-byte unsigned binary numbers and sets the Carry and Zero flags appropriately. The Zero flag is set to 1 if the operands are equal and to 0 if they are not equal. The Carry flag is set to 0 if the operand with the address higher in the stack (the subtrahend) is larger than the other operand (the minuend); the Carry flag is set to 1 otherwise. Thus, the flags are set as if the subtrahend had been subtracted from the minuend.

Procedure: The program compares the operands one byte at a time, starting with the most significant bytes and continuing until it finds corresponding bytes that are not equal. If all the bytes are equal, it exits with the Zero flag set to 1. Note that the comparison works through the operands starting with the most significant bytes, whereas the subtraction (Subroutine 6G) starts with the least significant bytes.

Registers Used: All

Execution Time: 17 cycles per byte that must be compared plus 90 cycles overhead. That is, the program continues until it finds corresponding bytes that are not equal; each pair of bytes it must examine requires 17 cycles.

Examples:

1. Comparing two 6-byte numbers that are equal
   $17 \times 6 + 90 = 192$ cycles
2. Comparing two 8-byte numbers that differ in the next to most significant bytes
   $17 \times 2 + 90 = 124$ cycles

Program Size: 54 bytes

Data Memory Required: Two bytes anywhere in RAM and four bytes on page 0. The two bytes anywhere in RAM are temporary storage for the return address (starting at address RETADR). The four bytes on page 0 hold pointers to the two numbers; the pointers start at addresses MINPTR ($00D0_{16}$ in the listing) and SUBPTR ($00D2_{16}$ in the listing).

Special Case: A length of zero causes an immediate exit with the Carry flag and the Zero flag both set to 1.

Entry Conditions

Order in stack (starting from top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of subtrahend (address containing the least significant byte)

More significant byte of starting address of subtrahend (address containing the least significant byte)

Less significant byte of starting address of minuend (address containing the least sig-

Exit Conditions

Flags set as if subtrahend had been subtracted from minuend

Zero flag = 1 if subtrahend and minuend are equal, 0 if they are not equal

Carry flag = 0 if subtrahend is larger than minuend in the unsigned sense, 1 if it is less than or equal to the minuend.

nificant byte)
More significant byte of starting address of minuend (address containing the least significant byte)

Examples

1. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = $19D028A193EA_{16}$
   Bottom operand (minuend) = $4E67BC15A266_{16}$

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 1 (subtrahend is not larger than minuend)

2. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = $19D028A193EA_{16}$
   Bottom operand (minuend) = $19D028A193EA_{16}$

   Result: Zero flag = 1 (operands are equal)
   Carry flag = 1 (subtrahend is not larger than minuend)

3. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = $19D028A193EA_{16}$
   Bottom operand (minuend) = $0F37E5991D7C_{16}$

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 0 (subtrahend is larger than minuend)

```
;       Title           Multiple-Precision Binary Comparision          ;
;       Name:           MPBCMP                                         ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;       Purpose:        Compare 2 arrays of binary bytes and return    ;
;                       the CARRY and ZERO flags set or cleared        ;
;                                                                      ;
;       Entry:          TOP OF STACK                                   ;
;                         Low byte of return address,                  ;
;                         High byte of return address,                 ;
;                         Length of the arrays in bytes,               ;
;                         Low byte of array 2 (subtrahend) address,    ;
;                         High byte of array 2 (subtrahend) address,   ;
;                         Low byte of array 1 (minuend) address,       ;
;                         High byte of array 1 (minuend) address       ;
;                                                                      ;
```

```
;                       The arrays are unsigned binary numbers with a ;
;                       maximum length of 255 bytes, ARRAY[0] is the  ;
;                       least significant byte, and ARRAY[LENGTH-1]   ;
;                       the most significant byte.                    ;
;                                                                     ;
;       Exit:          IF ARRAY 1 = ARRAY 2 THEN                      ;
;                        C=1,Z=1                                      ;
;                      IF ARRAY 1 > ARRAY 2 THEN                      ;
;                        C=1,Z=0                                      ;
;                      IF ARRAY 1 < ARRAY 2 THEN                      ;
;                        C=0,Z=0                                      ;
;                                                                     ;
;       Registers used: All                                           ;
;                                                                     ;
;       Time:          17 cycles per byte that must be examined       ;
;                      plus 90 cycles overhead.                       ;
;                                                                     ;
;       Size:          Program 54 bytes                               ;
;                      Data     2 bytes plus                          ;
;                               4 bytes in page zero                  ;
;                                                                     ;
;                                                                     ;

;EQUATES
MINPTR: .EQU    0D0H            ;PAGE ZERO FOR ARRAY 1 POINTER
SUBPTR: .EQU    0D2H            ;PAGE ZERO FOR ARRAY 2 POINTER

MPBCMP:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1        ;SAVE RETURN ADDRESS

        ;GET LENGTH OF ARRAYS
        PLA
        TAY

        ;GET ADDRESS OF SUBTRAHEND AND SUBTRACT 1 TO SIMPLIFY INDEXING
        PLA
        SEC
        SBC     #1              ;SUBTRACT 1 FROM LOW BYTE
        STA     SUBPTR
        PLA
        SBC     #0              ;SUBTRACT ANY BORROW FROM HIGH BYTE
        STA     SUBPTR+1

        ;GET ADDRESS OF MINUEND AND ALSO SUBTRACT 1
        PLA
        SEC
        SBC     #1              ;SUBTRACT 1 FROM LOW BYTE
        STA     MINPTR
        PLA
        SBC     #0              ;SUBTRACT ANY BORROW FROM HIGH BYTE
        STA     MINPTR+1
```

```
        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE
        CPY     #0              ;IS LENGTH OF ARRAYS = 0 ?
        BEQ     EXIT            ;YES, EXIT WITH C=1,Z=1

LOOP:
        LDA     (MINPTR),Y      ;GET NEXT BYTE
        CMP     (SUBPTR),Y      ;COMPARE BYTES
        BNE     EXIT            ;EXIT IF THEY ARE NOT EQUAL, THE FLAGS ARE SET
        DEY                     ;DECREMENT INDEX
        BNE     LOOP            ;CONTINUE UNTIL COUNTER = 0
                                ; IF WE FALL THROUGH THEN THE ARRAYS ARE EQUAL
                                ; AND THE FLAGS ARE SET PROPERLY

EXIT:
        RTS

;
;DATA
RETADR  .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS

;                                                                      ;
;                                                                      ;
;       SAMPLE EXECUTION:                                              ;
;                                                                      ;
;                                                                      ;
;

SC0610:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                     ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                     ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                     ;PUSH SIZE OF ARRAYS
        JSR     MPBCMP          ;MULTIPLE-PRECISION BINARY COMPARISON
        BRK                     ;RESULT OF COMPARE(7654321H,1234567H) IS
                                ;  C=1,Z=0
        JMP     SC0610

SZAYS:  .EQU    7               ;SIZE OF ARRAYS

AY1ADR: .WORD   AY1             ;ADDRESS OF ARRAY 1 (MINUEND)
AY2ADR: .WORD   AY2             ;ADDRESS OF ARRAY 2 (SUBTRAHEND)

AY1:
```

```
        .BYTE   021H
        .BYTE   043H
        .BYTE   065H
        .BYTE   007H
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   067H
        .BYTE   045H
        .BYTE   023H
        .BYTE   001H
        .BYTE   0
        .BYTE   0
        .BYTE   0

        .END    ;PROGRAM
```

Multiple-Precision Decimal Addition (MPDADD)

6K

Adds two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant digits at the lowest address. The sum replaces one of the numbers (the one with the starting address lower in the stack). The length of the numbers (in bytes) is 255 or less. The program returns with the Decimal Mode (D) flag cleared (binary mode).

Procedure: The program first enters the decimal mode by setting the D flag. It then clears the Carry flag initially and adds the operands one byte (two digits) at a time, starting with the least significant digits. The sum replaces the operand with the starting address lower in the stack (array 1 in the listing). A length of 00 causes an immediate exit with no addition operations. The program clears the D flag (thus placing the processor in the binary mode) before exiting. The final Carry flag reflects the addition of the most significant digits.

Registers Used: All

Execution Time: 23 cycles per byte plus 82 cycles overhead. For example, adding two 8-byte (16-digit) operands takes $23 \times 8 + 86$ or 270 cycles.

Program Size: 50 bytes

Data Memory Required: Two bytes anywhere in RAM and four bytes on page 0. The two bytes anywhere in RAM are temporary storage for the return address (starting at address RETADR). The four bytes on page 0 hold pointers to the two operands; the pointers start at addresses AY1PTR ($00D0_{16}$ in the listing) and AY2PTR ($00D2_{16}$ in the listing).

Special Case: A length of zero causes an immediate exit with array 1 unchanged (that is, the sum is equal to bottom operand). The Decimal Mode flag is cleared (binary mode) and the Carry flag is set to 1.

Entry Conditions

Order in stack (starting from top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of second operand (address containing the least significant byte of array 2)

More significant byte of starting address of second operand (address containing the least significant byte of array 2)

Less significant byte of starting address of first operand and result (address containing the least significant byte of array 1)

Exit Conditions

First operand (array 1) replaced by first operand (array 1) plus second operand (array 2).

D flag set to zero (binary mode).

More significant byte of starting address of first operand and result (address containing the least significant byte of array 1)

Example

Data: Length of operands (in bytes) = 6
Top operand (array 2) = 196028819315_{16}
Bottom operand (array 1) = 293471605987_{16}

Result: Bottom operand (array 1) = Bottom operand (array 1) + Top operand (array 2) = 489500425302_{16}
Carry = 0, Decimal Mode flag = 0 (binary mode)

```
;       Title               Multiple-Precision Decimal Addition      ;
;       Name:               MPDADD                                   ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Purpose:            Add 2 arrays of BCD bytes                ;
;                           Array1 := Array1 + Array2                ;
;                                                                    ;
;       Entry:              TOP OF STACK                             ;
;                             Low byte of return address,            ;
;                             High byte of return address,           ;
;                             Length of the arrays in bytes,         ;
;                             Low byte of array 2 address,           ;
;                             High byte of array 2 address,          ;
;                             Low byte of array 1 address,           ;
;                             High byte of array 1 address           ;
;                                                                    ;
;                             The arrays are unsigned BCD numbers with a ;
;                             maximum length of 255 bytes, ARRAY[0] is the ;
;                             least significant byte, and ARRAY[LENGTH-1] ;
;                             the most significant byte.             ;
;                                                                    ;
;       Exit:               Array1 := Array1 + Array2                ;
;                                                                    ;
;       Registers used: All                                          ;
;                                                                    ;
;       Time:               23 cycles per byte plus 86 cycles        ;
;                           overhead.                                ;
;                                                                    ;
```

```
;       Size:              Program 50 bytes                                          ;
;                          Data     2 bytes plus                                     ;
;                                   4 bytes in page zero                             ;
;                                                                                    ;
;                                                                                    ;

;EQUATES
AY1PTR: .EQU    0D0H            ;PAGE ZERO FOR ARRAY 1 POINTER
AY2PTR: .EQU    0D2H            ;PAGE ZERO FOR ARRAY 2 POINTER

MPDADD:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET LENGTH OF ARRAYS
        PLA
        TAX

        ;GET STARTING ADDRESS OF ARRAY 2
        PLA
        STA     AY2PTR
        PLA
        STA     AY2PTR+1

        ;GET STARTING ADDRESS OF ARRAY 1
        PLA
        STA     AY1PTR
        PLA
        STA     AY1PTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE SUM AND DECIMAL MODE, EXIT IF LENGTH = 0
        LDY     #0
        CPX     #0              ;IS LENGTH OF ARRAYS = 0 ?
        BEQ     EXIT            ;BRANCH IF LENGTH IS 0
        SED                     ;SET DECIMAL MODE
        CLC                     ;CLEAR CARRY

LOOP:
        LDA     (AY1PTR),Y      ;GET NEXT BYTE
        ADC     (AY2PTR),Y      ;ADD BYTES
        STA     (AY1PTR),Y      ;STORE SUM
        INY                     ;INCREMENT ARRAY INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE UNTIL COUNTER = 0

EXIT:
        CLD                     ;RETURN IN BINARY MODE
```

```
        RTS

;
;DATA
RETADR  .BLOCK  2               ;TEMPORARY FOR RETURN ADDRESS

;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;

SC0611:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                     ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                     ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                     ;PUSH SIZE OF ARRAYS
        JSR     MPDADD          ;MULTIPLE-PRECISION BCD ADDITION
        BRK                     ;RESULT OF 1234567 + 1234567 = 2469134
                                ; IN MEMORY AY1      = 34H
                                ;           AY1+1    = 91H
                                ;           AY1+2    = 46H
                                ;           AY1+3    = 02H
                                ;           AY1+4    = 00H
                                ;           AY1+5    = 00H
                                ;           AY1+6    = 00H
        JMP     SC0611

SZAYS:  .EQU    7               ;SIZE OF ARRAYS

AY1ADR: .WORD   AY1             ;ADDRESS OF ARRAY 1
AY2ADR: .WORD   AY2             ;ADDRESS OF ARRAY 2

AY1:
        .BYTE   067H
        .BYTE   045H
        .BYTE   023H
        .BYTE   001H
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   067H
        .BYTE   045H
        .BYTE   023H
```

```
.BYTE    001H
.BYTE    0
.BYTE    0
.BYTE    0

.END     ;PROGRAM
```

Multiple-Precision Decimal Subtraction
(MPDSUB)

6L

Subtracts two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant digits at the lowest address. The starting address of the subtrahend (number to be subtracted) is stored on top of the starting address of the minuend (number from which the subtrahend is subtracted). The difference replaces the minuend in memory. The length of the numbers (in bytes) is 255 or less. The program returns with the Decimal Mode (D) flag cleared (binary mode).

Procedure: The program first enters the decimal mode by setting the D flag. It then sets the Carry flag (the inverted borrow) initially and subtracts the subtrahend from the minuend one byte (two digits) at a time, starting with the least significant digits. The final Carry flag reflects the subtraction of the most significant digits. The difference replaces the minuend (the operand with the starting address lower in the stack, array 1 in the listing). A length of 00 causes an immediate exit with no subtraction operations. The program clears the D flag (thus placing the processor in the binary mode) before exiting.

Registers Used: All

Execution Time: 23 cycles per byte plus 86 cycles overhead. For example, subtracting two 8-byte (16-digit) operands takes 23 × 8 + 86 or 270 cycles.

Program Size: 50 bytes

Data Memory Required: Two bytes anywhere in RAM and four bytes on page 0. The two bytes anywhere in RAM are temporary storage for the return address (starting at address RETADR). The four bytes on page 0 hold pointers to the two operands; the pointers start at addresses AY1PTR ($00D0_{16}$ in the listing) and AY2PTR ($00D2_{16}$ in the listing).

Special Case: A length of zero causes an immediate exit with the difference equal to the original minuend, the Decimal Mode flag cleared (binary mode), and the Carry flag set to 1.

Entry Conditions
Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of subtrahend (address containing the least significant byte of array 2)
More significant byte of starting address of subtrahend (address containing the least significant byte of array 2)

Less significant byte of starting address of

Exit Conditions

Minuend (array 1) replaced by minuend (array 1) minus subtrahend (array 2).

D flag set to zero (binary mode).

minuend (address containing the least significant byte of array 1)

More significant byte of starting address of minuend (address containing the least significant byte of array 1)

Example

Data: Length of operands (in bytes) = 6

Minuend (array 1) = 293471605987_{16}

Subtrahend (array 2) = 196028819315_{16}

Result: Difference (array 1) = 097442786672_{16}.
This number replaces the original minuend in memory. The Carry flag is set to 1 in accordance with its usual role (in 6502 programming) as an inverted borrow.
Decimal Mode flag = 0 (binary mode)

```
;       Title          Multiple-Precision Decimal Subtraction          ;
;       Name:          MPDSUB                                          ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;       Purpose:       Subtract 2 arrays of BCD bytes                  ;
;                      Minuend := Minuend - Subtrahend                 ;
;                                                                      ;
;       Entry:         TOP OF STACK                                    ;
;                        Low byte of return address,                   ;
;                        High byte of return address,                  ;
;                        Length of the arrays in bytes,                ;
;                        Low byte of subtrahend address,               ;
;                        High byte of subtrahend address,              ;
;                        Low byte of minuend address,                  ;
;                        High byte of minuend address                  ;
;                                                                      ;
;                        The arrays are unsigned BCD numbers with a    ;
;                        maximum length of 255 bytes, ARRAY[0] is the  ;
;                        least significant byte, and ARRAY[LENGTH-1]   ;
;                        the most significant byte.                    ;
;                                                                      ;
;       Exit:          Array1 := Array1 - Array2                       ;
;                                                                      ;
;       Registers used: All                                            ;
```

```
;                                                                    ;
;       Time:               23 cycles per byte plus 86 cycles        ;
;                           overhead.                                ;
;                                                                    ;
;       Size:               Program 50 bytes                         ;
;                           Data     2 bytes plus                    ;
;                                    4 bytes in page zero            ;
;                                                                    ;
;                                                                    ;

;EQUATES
MINPTR: .EQU    0D0H                ;PAGE ZERO FOR MINUEND POINTER
SUBPTR: .EQU    0D2H                ;PAGE ZERO FOR SUBTRAHEND POINTER

MPDSUB:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET LENGTH OF ARRAYS
        PLA
        TAX

        ;GET STARTING ADDRESS OF SUBTRAHEND
        PLA
        STA     SUBPTR
        PLA
        STA     SUBPTR+1

        ;GET STARTING ADDRESS OF MINUEND
        PLA
        STA     MINPTR
        PLA
        STA     MINPTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE
        LDY     #0
        CPX     #0              ;IS LENGTH OF ARRAYS = 0 ?
        BEQ     EXIT            ;YES, EXIT
        SED                     ;SET DECIMAL MODE
        SEC                     ;SET CARRY
LOOP:
        LDA     (MINPTR),Y      ;GET NEXT BYTE
        SBC     (SUBPTR),Y      ;SUBTRACT BYTES
        STA     (MINPTR),Y      ;STORE DIFFERENCE
        INY                     ;INCREMENT ARRAY INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE UNTIL COUNTER = 0
```

```
EXIT:
        CLD                             ;RETURN IN BINARY MODE
        RTS

;
;DATA
RETADR  .BLOCK  2                       ;TEMPORARY FOR RETURN ADDRESS

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0612:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                             ;PUSH AY1 ADDRESS
        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                             ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                             ;PUSH SIZE OF ARRAYS
        JSR     MPDSUB                  ;MULTIPLE-PRECISION BCD SUBTRACTION
        BRK                             ;RESULT OF 2469134 - 1234567 = 1234567
                                        ; IN MEMORY AY1     = 67H
                                        ;           AY1+1   = 45H
                                        ;           AY1+2   = 23H
                                        ;           AY1+3   = 01H
                                        ;           AY1+4   = 00H
                                        ;           AY1+5   = 00H
                                        ;           AY1+6   = 00H
        JMP     SC0612

SZAYS:  .EQU    7                       ;SIZE OF ARRAYS

AY1ADR: .WORD   AY1                     ;ADDRESS OF ARRAY 1 (MINUEND)
AY2ADR: .WORD   AY2                     ;ADDRESS OF ARRAY 2 (SUBTRAHEND)

AY1:
        .BYTE   034H
        .BYTE   091H
        .BYTE   046H
        .BYTE   002H
        .BYTE   0
        .BYTE   0
        .BYTE   0
```

```
AY2:
        .BYTE   067H
        .BYTE   045H
        .BYTE   023H
        .BYTE   001H
        .BYTE   0
        .BYTE   0
        .BYTE   0

        .END    ;PROGRAM
```

Multiple-Precision Decimal Multiplication (MPDMUL) 6M

Multiplies two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant digits at the lowest address. The product replaces one of the numbers (the one with the starting address lower in the stack). The length of the numbers (in bytes) is 255 or less. Only the least significant bytes of the product are returned to retain compatibility with other multiple-precision decimal operations. The program returns with the Decimal Mode (D) flag cleared (binary mode).

Procedure: The program handles each digit of the multiplicand (array 1) separately. It masks that digit off, shifts it (if it is in the upper nibble of a byte), and then uses it as a counter to determine how many times to add the multiplier to the partial product. The least significant digit of the partial product is saved as the next digit of the full product and the partial product is shifted right four bits. The program uses a flag to determine whether it is currently working with the upper or lower digit of a byte. A length of 00 causes an exit with no multiplication.

Registers Used: All

Execution Time: Depends on the length of the operands and on the size of the digits in the multiplicand (since those digits determine how many times the multiplier is added to the partial product).

If the average digit in the multiplicand has a value of 5, then the execution time is approximately

$$322 \times \text{LENGTH}^2 + 390 \times \text{LENGTH} + 100$$

cycles where LENGTH is the number of bytes in the operand. If, for example, LENGTH = 6 (12 digits), the approximate execution time is

$$322 \times 6^2 + 390 \times 6 + 100 = 322 \times 36 + 2340 + 100 = 11{,}592 + 2440 = 14{,}032 \text{ cycles.}$$

Program Size: 203 bytes

Data Memory Required: 517 bytes anywhere in RAM plus four bytes on page 0. The 517 bytes anywhere in RAM are temporary storage for the partial product (255 bytes starting at address PROD), the multiplicand (255 bytes starting at address MCAND), the return address (two bytes starting at address RETADR), the length of the operands in bytes (one byte at address LENGTH), the next digit in the operand (one byte at address NDIGIT), the digit counter (one byte at address DCNT), the byte index into the operands (one byte at address IDX), and the overflow byte (1 byte at address OVERFLW). The four bytes on page 0 hold pointers to the two operands; the pointers start at addresses AY1PTR ($00D0_{16}$ in the listing) and AY2PTR ($00D2_{16}$ in the listing).

Special Case: A length of zero causes an immediate exit with the product equal to the original multiplicand (array 1 is unchanged), the Decimal Mode flag cleared (binary mode), and the more significant bytes of the product (starting at address PROD) undefined.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Exit Conditions

Multiplicand (array 1) replaced by multiplicand (array 1) times multiplier (array 2).

D flag set to zero (binary mode).

Less significant byte of starting address of multiplier (address containing the least significant byte of array 2)

More significant byte of starting address of multiplier (address containing the least significant byte of array 2)

Less significant byte of starting address of multiplicand (address containing the least significant byte of array 1)

More significant byte of starting address of multiplicand (address containing the least significant byte of array 1)

Example

Data: Length of operands (in bytes) = 04

Top operand (array 2 or multiplier) = 00003518_{16}

Bottom operand (array 1 or multiplicand) = 00006294_{16}

Result: Bottom operand (array 1) = Bottom operand (array 1) * Top operand (array 2) = 22142292_{16}.

Decimal Mode flag = 0 (binary mode)

Note that MPDMUL returns only the less significant bytes of the product (that is, the number of bytes in the multiplicand and multiplier) to maintain compatibility with other multiple-precision decimal arithmetic operations. The more significant bytes of the product are available starting with their least significant digits at address PROD. The user may need to check those bytes for a possible overflow or extend the operands with additional zeros.

```
;       Title            Multiple-Precision Decimal Multiplication        ;
;       Name:            MPDMUL                                           ;
;                                                                         ;
;       Purpose:         Multiply 2 arrays of BCD bytes                   ;
;                        Arrayl := Arrayl * Array2                        ;
;                                                                         ;
;       Entry:           TOP OF STACK                                     ;
;                          Low byte of return address,                    ;
;                          High byte of return address,                   ;
;                          Length of the arrays in bytes,                 ;
;                          Low byte of array 2 (mulitplicand) address,    ;
;                          High byte of array 2 (multiplicand) address,   ;
;                          Low byte of array 1 (multiplier) address,      ;
;                          High byte of array 1 (multiplier) address      ;
;                                                                         ;
;                          The arrays are unsigned BCD numbers with a     ;
;                          maximum length of 255 bytes, ARRAY[0] is the   ;
;                          least significant byte, and ARRAY[LENGTH-1]    ;
;                          the most significant byte.                     ;
;                                                                         ;
;       Exit:            Arrayl := Arrayl * Array2                        ;
```

```
;                                                                         ;
;       Registers used: All                                               ;
;                                                                         ;
;       Time:           Assuming the average digit value of ARRAY 1 is    ;
;                       5 then the time is approximately                  ;
;                        (322 * length^2) + (390 * length) + 100 cycles   ;
;                                                                         ;
;       Size:           Program 203 bytes                                 ;
;                       Data    517 bytes plus                            ;
;                                 4 bytes in page zero                    ;
;                                                                         ;
;                                                                         ;
;

;EQUATES
AY1PTR: .EQU    0D0H            ;PAGE ZERO FOR ARRAY 1 POINTER
AY2PTR: .EQU    0D2H            ;PAGE ZERO FOR ARRAY 2 POINTER

MPDMUL:
        ;SAVE RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET LENGTH OF ARRAYS
        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF ARRAY 2
        PLA
        STA     AY2PTR
        PLA
        STA     AY2PTR+1

        ;GET STARTING ADDRESS OF ARRAY 1
        PLA
        STA     AY1PTR
        PLA
        STA     AY1PTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE
        SED                     ;PUT PROCESSOR IN DECIMAL MODE
        LDY     #0
        LDX     LENGTH          ;IS LENGTH ZERO ?
        BNE     INITLP
        JMP     EXIT            ;YES, EXIT

        ;MOVE ARRAY 1 TO MULTIPLICAND ARRAY, ZERO ARRAY 1, AND
        ; ZERO PRODUCT ARRAY.
INITLP:
        LDA     (AY1PTR),Y
        STA     MCAND,Y         ;MOVE ARY1[Y] TO MCAND[Y]
```

```
        LDA     #0
        STA     (AY1PTR),Y      ;ZERO ARY1[Y]
        STA     PROD,Y          ;ZERO PROD
        INY
        DEX                     ;DECREMENT LOOP COUNTER
        BNE     INITLP          ;CONTINUE UNTIL DONE

        ;INITIALIZE CURRENT INDEX TO ZERO
        LDA     #0
        STA     IDX

        ;
        ;LOOP THROUGH ALL THE BYTES OF THE MULTIPLICAND
LOOP:
        LDA     #0
        STA     DCNT            ;START WITH LOW DIGIT

        ;LOOP THROUGH 2 DIGITS PER BYTE
        ; DURING THE FIRST DIGIT DCNT = 0
        ; DURING THE SECOND DIGIT DCNT = FF HEX (-1)
DLOOP:
        LDA     #0
        STA     OVRFLW          ;ZERO OVERFLOW
        LDY     IDX
        LDA     MCAND,Y         ;GET NEXT BYTE
        LDX     DCNT
        BPL     DLOOP1          ;BRANCH IF FIRST DIGIT
        LSR     A               ;SHIFT RIGHT 4 BITS
        LSR     A
        LSR     A
        LSR     A
DLOOP1:
        AND     #0FH            ;AND OFF UPPER DIGIT
        BEQ     SDIGIT          ;BRANCH IF NEXT DIGIT IS ZERO
        STA     NDIGIT          ;SAVE

        ;ADD MULTIPLIER TO PRODUCT NDIGIT TIMES
ADDLP:
        LDY     #0              ;Y = INDEX INTO ARRAYS
        LDX     LENGTH          ;X = LENGTH IN BYTES
        CLC                     ;CLEAR CARRY INITIALY

INNER:
        LDA     (AY2PTR),Y      ;GET NEXT BYTE
        ADC     PROD,Y          ;ADD TO PRODUCT
        STA     PROD,Y          ;STORE
        INY                     ;INCREMENT ARRAY INDEX
        DEX                     ;DECREMENT LOOP COUNTER
        BNE     INNER           ;CONTINUE UNTIL LOOP COUNTER = 0
        BCC     DECND           ;BRANCH IF NO OVERFLOW FROM ADDITION
        INC     OVRFLW          ;ELSE INCREMENT OVERFLOW BYTE

DECND:
        DEC     NDIGIT
        BNE     ADDLP           ;CONTINUE UNTIL NDIGIT = 0
```

```
        ;STORE THE LEAST SIGNIFICANT DIGIT OF PRODUCT
        ; AS THE NEXT DIGIT OF ARRAY 1
SDIGIT:
        LDA     PROD
        AND     #0FH            ;CLEAR UPPER DIGIT
        LDX     DCNT
        BPL     SD1             ;BRANCH IF FIRST DIGIT
        ASL     A               ;ELSE SHIFT LEFT 4 BITS TO PLACE
        ASL     A               ; IN THE UPPER DIGIT
        ASL     A
        ASL     A
SD1:
        LDY     IDX             ;GET CURRENT BYTE INDEX
        ORA     (AY1PTR),Y      ;OR IN NEXT DIGIT
        STA     (AY1PTR),Y      ;STORE NEW VALUE

        ;SHIFT RIGHT PRODUCT 1 DIGIT (4 BITS)
        LDY     LENGTH          ;SHIFT RIGHT FROM THE FAR END

SHFTLP:
        DEY                     ;DECREMENT Y SO IT POINTS AT THE NEXT BYTE
        LDA     PROD,Y
        PHA                     ;SAVE LOW DIGIT OF PROD,Y
        AND     #0F0H           ;CLEAR LOW DIGIT

        ;MAKE LOW DIGIT OF OVERFLOW = HIGH DIGIT OF PROD,Y
        ;MAKE HIGH DIGIT OF PROD,Y = LOW DIGIT OF PROD,Y
        LSR     OVRFLW          ;SHIFT OVERFLOW RIGHT
        ORA     OVRFLW          ;BIT 0..2 AND CARRY = OVERFLOW
                                ;BITS 4..7 = PROD
        ROR     A
        ROR     A
        ROR     A
        ROR     A               ;NOW PROD IN BITS 0..3 AND OVERFLOW IN 4..7
        STA     PROD,Y          ;STORE NEW PRODUCT
        PLA                     ;GET OLD PROD,Y
        AND     #0FH            ;CLEAR UPPER DIGIT
        STA     OVRFLW          ;STORE IN OVERFLOW
        TYA                     ;CHECK FOR Y = 0
        BNE     SHFTLP          ;BRANCH IF NOT DONE

        ;CHECK IF WE ARE DONE WITH BOTH DIGITS OF THIS BYTE
        DEC     DCNT            ;MAKE 0 GOTO FF HEX TO INDICATE SECOND DIGIT
        LDA     DCNT
        CMP     #0FFH           ;HAVE WE ALREADY DONE BOTH DIGITS ?
        BEQ     DLOOP           ;BRANCH IF NOT

        ;INCREMENT TO NEXT BYTE AND SEE IF WE ARE DONE
        INC     IDX
        LDA     IDX
        CMP     LENGTH
        BCS     EXIT            ;BRANCH IF BYTE INDEX >= LENGTH
        JMP     LOOP            ;ELSE CONTINUE

EXIT:
```

```
        CLD                         ;RETURN IN BINARY MODE
        RTS

;
;DATA
RETADR:          .BLOCK   2         ;TEMPORARY FOR RETURN ADDRESS
LENGTH:          .BLOCK   1         ;LENGTH OF ARRAYS
NDIGIT:          .BLOCK   1         ;NEXT DIGIT IN ARRAY
DCNT:            .BLOCK   1         ;DIGIT COUNTER FOR BYTES IN ARRAYS
IDX:             .BLOCK   1         ;BYTE INDEX INTO ARRAYS
OVRFLW:          .BLOCK   1         ;OVERFLOW BYTE
PROD:            .BLOCK   255       ;PRODUCT BUFFER
MCAND:           .BLOCK   255       ;MULTIPLICAND BUFFER

;                                                                   ;
;                                                                   ;
;       SAMPLE EXECUTION:                                           ;
;                                                                   ;
;                                                                   ;

SC0613:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                         ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                         ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                         ;PUSH LENGTH OF ARRAYS
        JSR     MPDMUL              ;MULTIPLE-PRECISION BCD MULTIPLICATION
        BRK                         ;RESULT OF 1234 * 1234 = 1522756
                                    ; IN MEMORY AY1      = 56H
                                    ;           AY1+1    = 27H
                                    ;           AY1+2    = 52H
                                    ;           AY1+3    = 01H
                                    ;           AY1+4    = 00H
                                    ;           AY1+5    = 00H
                                    ;           AY1+6    = 00H
        JMP     SC0613

SZAYS:  .EQU    7                   ;LENGTH OF ARRAYS

AY1ADR: .WORD   AY1                 ;ADDRESS OF ARRAY 1
AY2ADR: .WORD   AY2                 ;ADDRESS OF ARRAY 2

AY1:
        .BYTE   034H
        .BYTE   012H
        .BYTE   0
        .BYTE   0
```

```
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   034H
        .BYTE   012H
        .BYTE   0
        .BYTE   0
        .BYTE   0
        .BYTE   0
        .BYTE   0

        .END    ;PROGRAM
```

Multiple-Precision Decimal Division (MPDDIV)

Divides two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant byte at the lowest address. The quotient replaces the dividend (the operand with the starting address lower in the stack). The length of the numbers (in bytes) is 255 or less. The remainder is not returned but the address of its least significant byte is available starting at memory location HDEPTR. The Carry flag is cleared if no errors occur; if a divide by zero is attempted, the Carry flag is set to 1, the dividend is left unchanged, and the remainder is set to zero. The program returns with the Decimal Mode (D) flag cleared (binary mode).

Procedure: The program performs division by trial subtractions, a digit at a time. It determines how many times the divisor can be subtracted from the dividend and then saves that number in the quotient and makes the remainder into the new dividend. It then rotates the dividend and the quotient left one digit. The program exits immediately, setting the Carry flag, if it finds the divisor to be zero. The Carry flag is cleared otherwise.

Registers Used: All

Execution Time: Depends on the length of the operands and on the size of the digits in the quotient (determining how many trial subtractions must be performed). If the average digit in the quotient has a value of 5, then the execution time is approximately

$$440 \times \text{LENGTH}^2 + 765 \times \text{LENGTH} + 228$$

cycles where LENGTH is the number of bytes in the operands. If, for example, LENGTH = 6 (12 digits), the approximate execution time is

$$440 \times 6^2 + 765 \times 6 + 228 = 440 \times 36 + 4590 + 228 = 15{,}840 + 4818 = 20{,}658 \text{ cycles.}$$

Program Size: 246 bytes

Data Memory Required: 522 bytes anywhere in RAM plus eight bytes on page 0. The 522 bytes anywhere in RAM are temporary storage for the high dividend (255 bytes starting at address HIDE1), the result of the trial subtraction (255 bytes starting at address HIDE2), the return address (two bytes starting at address RETADR), a pointer to the dividend (two bytes starting at address AY1PTR), the length of the operands (one byte at address LENGTH), the next digit in the array (one byte at address NDIGIT), the divide loop counter (one byte at address COUNT), and the addresses of the high dividend buffers (two bytes each, starting at addresses AHIDE1 and AHIDE2). The eight bytes on page 0 hold pointers to the divisor (address AY2PTR, $00D0_{16}$ in the listing), the current high dividend and remainder (address HDEPTR, $00D2_{16}$ in the listing), the other high dividend (address ODEPTR, $00D4_{16}$ in the listing), and the temporary array used in the left rotation (address RLPTR, $00D6_{16}$ in the listing).

Special Cases:

1. A length of zero causes an immediate exit with the Carry flag cleared, the quotient equal to the original dividend (array 1 unchanged), the remainder undefined, and the Decimal Mode flag cleared (binary mode).

2. A divisor of zero causes an exit with the Carry flag set to 1, the quotient equal to the original dividend (array 1 unchanged), the remainder equal to zero, and the Decimal Mode flag cleared (binary mode).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length of the operands in bytes

Less significant byte of starting address of divisor (address containing the least significant byte of array 2)
More significant byte of starting address of divisor (address containing the least significant byte of array 2)

Less significant byte of starting address of dividend (address containing the least significant byte of array 1)
More significant byte of starting address of dividend (address containing the least significant byte of array 1)

Exit Conditions

Dividend (array 1) replaced by dividend (array 1) divided by divisor (array 2)

If the divisor is non-zero, Carry = 0 and the result is normal.

If the divisor is zero, Carry = 1, the dividend is unchanged, and the remainder is zero.

The remainder is available with its least significant digits stored at the address in HDEPTR and HDEPTR+1

D flag set to zero (binary mode).

Example

Data: Length of operands (in bytes) = 04

Top operand (array 2 or divisor) = 00006294_{16}

Bottom operand (array 1 or dividend) = 22142298_{16}

Result: Bottom operand (array 1) = Bottom operand (array 1)/Top operand (array 2) = 00003518_{16}

Remainder (starting at address in HDEPTR and HDEPTR+1) = $00000006_{16} = 6_{10}$

Decimal Mode flag = 0 (binary mode)

Carry flag is 0 to indicate no divide by zero error.

```
;       Title               Multiple-Precision Decimal Division          ;
;       Name:               MPDDIV                                       ;
;                                                                        ;
;                                                                        ;
;                                                                        ;
;       Purpose:            Divide 2 arrays of BCD bytes                 ;
;                           Arrayl := Arrayl / Array2                    ;
;                                                                        ;
;       Entry:              TOP OF STACK                                 ;
;                             Low byte of return address,                ;
;                             High byte of return address,               ;
;                             Length of the arrays in bytes,             ;
;                             Low byte of array 2 (divisor) address,     ;
;                             High byte of array 2 (divisor) address,    ;
;                             Low byte of array 1 (dividend) address,    ;
;                             High byte of array 1 (dividend)  address   ;
;                                                                        ;
;                             The arrays are unsigned BCD numbers with a ;
;                             maximum length of 255 bytes, ARRAY[0] is the ;
;                             least significant byte, and ARRAY[LENGTH-1] ;
;                             the most significant byte.                 ;
;                                                                        ;
;       Exit:               Arrayl := Arrayl / Array2                    ;
;                           Dvbuf := remainder                           ;
;                           If no errors then                            ;
;                             carry := 0                                 ;
;                           ELSE                                         ;
;                             divide by 0 error                          ;
;                             carry := 1                                 ;
;                             ARRAY 1 := unchanged                       ;
;                             remainder := 0                             ;
;                                                                        ;
;       Registers used: All                                              ;
;                                                                        ;
;       Time:               Assuming the average digit value in the      ;
;                           quotient is 5 then the time is approximately ;
;                            (440 * length^2) + (765 * length) + 228 cycles ;
;                                                                        ;
;       Size:               Program 246 bytes                            ;
;                           Data    522 bytes plus                       ;
;                                     8 bytes in page zero               ;
;                                                                        ;
;                                                                        ;

;EQUATES
AY2PTR: .EQU    0D0H            ;PAGE ZERO FOR ARRAY 2 (DIVISOR) POINTER
HDEPTR: .EQU    0D2H            ;PAGE ZERO WHICH POINTS TO THE CURRENT
                                ; HIGH DIVIDEND POINTER
ODEPTR: .EQU    0D4H            ;PAGE ZERO WHICH POINTS TO THE OTHER
                                ; HIGH DIVIDEND POINTER
RLPTR:  .EQU    0D6H            ;PAGE ZERO FOR ROTATE LEFT ARRAY

MPDDIV:
```

```
        ;GET RETURN ADDRESS
        PLA
        STA     RETADR
        PLA
        STA     RETADR+1

        ;GET LENGTH OF ARRAYS
        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF DIVISOR
        PLA
        STA     AY2PTR
        PLA
        STA     AY2PTR+1

        ;GET STARTING ADDRESS OF DIVIDEND
        PLA
        STA     AY1PTR
        PLA
        STA     AY1PTR+1

        ;RESTORE RETURN ADDRESS
        LDA     RETADR+1
        PHA
        LDA     RETADR
        PHA

        ;INITIALIZE
        CLD                     ;PUT PROCESSOR INTO BINARY MODE

        ;CHECK FOR ZERO LENGTH ARRAYS
        LDA     LENGTH
        BNE     INIT            ;BRANCH IF NOT ZERO
        JMP     OKEXIT          ;ELSE EXIT

        ;ZERO BOTH DIVIDEND BUFFERS
INIT:
        LDA     #0              ;A = 0
        LDY     LENGTH          ;X = LENGTH
INITLP:
        STA     HIDE1-1,Y
        STA     HIDE2-1,Y
        DEY
        BNE     INITLP

        ;SET UP THE HIGH DIVIDEND POINTERS
        LDA     AHIDE1
        STA     HDEPTR
        LDA     AHIDE1+1
        STA     HDEPTR+1
        LDA     AHIDE2
        STA     ODEPTR
        LDA     AHIDE2+1
        STA     ODEPTR+1
```

```
        ;NDIGIT := 0
        LDA     #0
        STA     NDIGIT

        ;SET COUNT TO NUMBER OF DIGITS PLUS 1
        ; COUNT := (LENGTH * 2) + 1
        LDA     LENGTH
        ASL     A               ;LENGTH * 2
        STA     COUNT
        LDA     #0
        ROL     A               ;MOVE OVERFLOW FROM * 2 INTO A
        STA     COUNT+1         ;STORE HIGH BYTE OF COUNT
        INC     COUNT
        BNE     CHKDV0          ;BRANCH IF NO OVERFLOW
        INC     COUNT+1

        ;CHECK FOR DIVIDE BY ZERO
CHKDV0:
        LDX     LENGTH
        LDY     #0
        TYA
DV01:
        ORA     (AY2PTR),Y
        INY
        DEX
        BNE     DV01            ;CONTINUE ORING ALL THE BYTES

        CMP     #0
        BNE     DVLOOP          ;BRANCH IF DIVISOR IS NOT 0
        JMP     EREXIT          ;ERROR EXIT

        ;PERFORM DIVISION BY TRIAL SUBTRACTIONS
DVLOOP:
        ;ROTATE LEFT THE LOWER DIVIDEND AND THE QUOTIENT (ARRAY 1)
        ; THE HIGH DIGIT OF NDIGIT BECOMES THE LEAST SIGNIFICANT DIGIT
        ; OF THE QUOTIENT (ARRAY 1) AND THE MOST SIGNIFICANT DIGIT
        ; OF ARRAY 1 (DIVIDEND) GOES TO THE HIGH DIGIT OF NDIGIT
        LDA     AY1PTR+1
        LDY     AY1PTR
        JSR     RLARY           ;ROTATE ARRAY 1

        ;IF COUNT = 0 THEN WE ARE DONE
        DEC     COUNT
        BNE     ROLDVB          ;BRANCH IF LOWER BYTE IS NOT 0
        LDA     COUNT+1         ;ELSE GET HIGH BYTE
        BEQ     OKEXIT          ;CONTINUE UNTIL COUNT = 0
        DEC     COUNT+1         ;DECREMENT UPPER BYTE OF COUNT

        ;

        ;ROTATE LEFT THE HIGH DIVIDEND WHERE THE LEAST SIGNIFICANT DIGIT
        ; OF HIGH DIVIDEND BECOMES THE HIGH DIGIT OF NDIGIT
ROLDVB:
        LDA     HDEPTR+1
        LDY     HDEPTR
        JSR     RLARY
```

```
        ;
        ;SEE HOW MANY TIMES THE DIVISOR WILL GO INTO THE HIGH DIVIDEND
        ; WHEN WE EXIT FROM THIS LOOP THE HIGH DIGIT OF NDIGIT IS THE NEXT
        ; QUOTIENT DIGIT AND HIGH DIVIDEND IS THE REMAINDER
        LDA     #0
        STA     NDIGIT          ;NDIGIT := 0
        SED                     ;ENTER DECIMAL MODE
SUBLP:
        LDY     #0              ;Y = INDEX INTO ARRAYS
        LDX     LENGTH          ;X = LENGTH
        SEC                     ;SET INVERTED BORROW

INNER:
        LDA     (HDEPTR),Y      ;GET NEXT BYTE OF DIVIDEND
        SBC     (AY2PTR),Y      ;SUBTRACT BYTE OF DIVISOR
        STA     (ODEPTR),Y      ;SAVE DIFFERENCE FOR NEXT SUBTRACTION
        INY                     ;INCREMENT ARRAY INDEX
        DEX                     ;DECREMENT LOOP COUNTER
        BNE     INNER           ;CONTINUE THROUGH ALL THE BYTES
        BCC     DVLOOP          ;BRANCH WHEN BORROW OCCURS AT WHICH TIME
                                ; NDIGIT IS THE NUMBER OF TIMES THE DIVISOR
                                ; GOES INTO THE ORIGINAL HIGH DIVIDEND AND
                                ; HIGH DIVIDEND CONTAINS THE REMAINDER.

        ;INCREMENT NEXT DIGIT WHICH IS IN THE HIGH DIGIT OF NDIGIT
        LDA     NDIGIT
        CLC
        ADC     #10H
        STA     NDIGIT

        ;EXCHANGE POINTERS, THUS MAKING REMAINDER THE NEW DIVIDEND
        LDX     HDEPTR
        LDY     HDEPTR+1
        LDA     ODEPTR
        STA     HDEPTR
        LDA     ODEPTR+1
        STA     HDEPTR+1
        STX     ODEPTR
        STY     ODEPTR+1

        JMP     SUBLP           ;CONTINUE UNTIL DIFFERENCE GOES NEGATIVE

        ;NO ERRORS, CLEAR CARRY
OKEXIT:
        CLC
        BCC     EXIT

        ;DIVIDE BY ZERO ERROR, SET CARRY

EREXIT:
        SEC

EXIT:
        ;HDEPTR CONTAINS THE ADDRESS OF THE REMAINDER
        CLD                     ;RETURN IN BINARY MODE
        RTS
```

```
;
;************************************
;SUBROUTINE: RLARY
;PURPOSE:    ROTATE LEFT AN ARRAY ONE DIGIT (4 BITS)
;ENTRY: A = HIGH BYTE OF ARRAY ADDRESS
;       Y = LOW BYTE OF ARRAY ADDRESS
;       THE HIGH DIGIT OF NDIGIT IS THE DIGIT TO ROTATE THROUGH
;EXIT:  ARRAY ROTATED LEFT THROUGH THE HIGH DIGIT OF NDIGIT
;REGISTERS USED: ALL
;************************************

RLARY:
        ;STORE ARRAY ADDRESS
        STA     RLPTR+1
        STY     RLPTR

        ;SHIFT NDIGIT INTO LOW DIGIT OF ARRAY AND
        ; SHIFT ARRAY LEFT
        LDX     LENGTH
        LDY     #0              ;START AT ARY1[0]

SHIFT:
        LDA     (RLPTR),Y       ;GET NEXT BYTE
        PHA                     ;SAVE HIGH DIGIT
        AND     #0FH            ;CLEAR HIGH DIGIT

        ASL     NDIGIT
        ORA     NDIGIT          ;BITS 0..3 = LOW DIGIT OF ARRAY
                                ;BITS 5..7 AND CARRY = NEXT DIGIT
        ROL     A
        ROL     A
        ROL     A
        ROL     A               ;NOW NDIGIT IN BITS 0..3 AND
                                ; LOW DIGIT IN HIGH DIGIT
        STA     (RLPTR),Y       ;STORE IT
        PLA                     ;GET OLD HIGH DIGIT
        AND     #0F0H           ;CLEAR LOWER DIGIT
        STA     NDIGIT          ;STORE IN NDIGIT
        INY                     ;INCREMENT TO NEXT BYTE
        DEX                     ;DECREMENT COUNT
        BNE     SHIFT           ;BRANCH IF NOT DONE

        RTS


;
;DATA
RETADR:         .BLOCK  2       ;TEMPORARY FOR RETURN ADDRESS
AY1PTR:         .BLOCK  2       ;ARRAY 1 ADDRESS
LENGTH:         .BLOCK  1       ;LENGTH OF ARRAYS
NDIGIT:         .BLOCK  1       ;NEXT DIGIT IN ARRAY
COUNT:          .BLOCK  2       ;DIVIDE LOOP COUNTER
AHIDE1:         .WORD   HIDE1   ;ADDRESS OF HIGH DIVIDEND BUFFER 1
AHIDE2:         .WORD   HIDE2   ;ADDRESS OF HIGH DIVIDEND BUFFER 2
HIDE1:          .BLOCK  255.    ;HIGH DIVIDEND BUFFER 1
HIDE2:          .BLOCK  255.    ;HIGH DIVIDEND BUFFER 2
```

```
;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC0614:
        LDA     AY1ADR+1
        PHA
        LDA     AY1ADR
        PHA                     ;PUSH AY1 ADDRESS

        LDA     AY2ADR+1
        PHA
        LDA     AY2ADR
        PHA                     ;PUSH AY2 ADDRESS

        LDA     #SZAYS
        PHA                     ;PUSH LENGTH OF ARRAYS
        JSR     MPDDIV          ;MULTIPLE-PRECISION BCD DIVISION
        BRK                     ;RESULT OF 1522756 / 1234 = 1234
                                ; IN MEMORY AY1      = 34H
                                ;           AY1+1    = 12H
                                ;           AY1+2    = 00H
                                ;           AY1+3    = 00H
                                ;           AY1+4    = 00H
                                ;           AY1+5    = 00H
                                ;           AY1+6    = 00H
        JMP     SC0614

SZAYS:  .EQU    7               ;LENGTH OF ARRAYS

AY1ADR: .WORD   AY1             ;ADDRESS OF ARRAY 1 (DIVIDEND)
AY2ADR: .WORD   AY2             ;ADDRESS OF ARRAY 2 (DIVISOR)

AY1:
        .BYTE   056H
        .BYTE   027H
        .BYTE   052H
        .BYTE   01H
        .BYTE   0
        .BYTE   0
        .BYTE   0

AY2:
        .BYTE   034H
        .BYTE   012H
        .BYTE   0
        .BYTE   0
        .BYTE   0
        .BYTE   0
        .BYTE   0

        .END    ;PROGRAM
```

Multiple-Precision Decimal Comparison (6O)

Compares two multi-byte unsigned decimal (BCD) numbers and sets the Carry and Zero flags appropriately. The Zero flag is set to 1 if the operands are equal and to 0 if they are not equal. The Carry flag is set to 0 if the operand with the address higher in the stack (the subtrahend) is larger then the other operand (the minuend); the Carry flag is set to 1 otherwise. Thus the flags are set as if the subtrahend had been subtracted from the minuend.

Note: This program is exactly the same as Subroutine 6J, the multiple-precision binary comparison, since the CMP instruction operates the same in the decimal mode as in the binary mode. Hence, see Subroutine 6J for a listing and other details.

Examples

1. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = 196528719340_{16}
   Bottom operand (minuend) = 456780153266_{16}

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 1 (subtrahend is not larger than minuend)

2. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = 196528719340_{16}
   Bottom operand (minuend) = 196528719340_{16}

   Result: Zero flag = 1 (operands are equal)
   Carry flag = 1 (subtrahend is not larger than minuend)

3. Data: Length of operands (in bytes) = 6
   Top operand (subtrahend) = 196528719340_{16}
   Bottom operand (minuend) = 073785991074_{16}

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 0 (subtrahend is larger than minuend)

Bit Set (BITSET)

Sets a specified bit in a 16-bit word to 1.

Procedure: The program uses bits 0 through 2 of register X to determine which bit position to set and bit 3 to select a particular byte of the original word-length data. It then logically ORs the selected byte with a mask containing a 1 in the chosen bit position and 0s elsewhere. The masks with one 1 bit are available in a table.

Registers Used: All

Execution Time: 57 cycles

Program Size: 42 bytes

Data Memory Required: Two bytes anywhere in RAM (starting at address VALUE).

Special Case: Bit positions above 15 will be interpreted mod 16. That is, for example, bit position 16 is equivalent to bit position 0.

Entry Conditions

More significant byte of data in accumulator

Less significant byte of data in register Y

Bit number to set in register X

Exit Conditions

More significant byte of result in accumulator

Less significant byte of result in register Y

Examples

1. Data: (A) = $6E_{16}$ = 01101110_2 (more significant byte)
   (Y) = 39_{16} = 00111001_2 (less significant byte)
   (X) = $0C_{16}$ = 12_{10} (bit position to set)

   Result: (A) = $7E_{16}$ = 01111110_2 (more significant byte, bit 12 set to 1)
   (Y) = 39_{16} = 00111001_2 (less significant byte)

2. Data: (A) = $6E_{16}$ = 01101110_2 (more significant byte)
   (Y) = 39_{16} = 00111001_2 (less significant byte)
   (X) = 02_{16} = 2_{10} (bit position to set)

   Result: (A) = $6E_{16}$ = 01101110_2 (more significant byte)
   (Y) = $3D_{16}$ = 00111101_2 (less significant byte, bit 2 set to 1)

```
;                                                                         ;
;       Title               Bit set                                       ;
;       Name:               BITSET                                        ;
;                                                                         ;
;                                                                         ;
;                                                                         ;
;       Purpose:            Set a bit in a 16 bit word.                   ;
;                                                                         ;
;       Entry:              Register A = High byte of word                ;
;                           Register Y = Low byte of word                 ;
;                           Register X = Bit number to set                ;
;                                                                         ;
;       Exit:               Register A = High byte of word with bit set   ;
;                           Register Y = Low byte of word with bit set    ;
;                                                                         ;
;       Registers used: All                                               ;
;                                                                         ;
;       Time:               57 cycles                                     ;
;                                                                         ;
;       Size:               Program 42 bytes                              ;
;                           Data     2 bytes                              ;
;                                                                         ;
;                                                                         ;

BITSET:
        ;SAVE THE DATA WORD
        STA     VALUE+1
        STY     VALUE

        ;BE SURE THAT THE BIT NUMBER IS BETWEEN 0 AND 15
        TXA
        AND     #0FH

        ;DETERMINE WHICH BYTE AND WHICH BIT IN THAT BYTE
        TAX                         ;SAVE BIT NUMBER IN X
        AND     #07H                ;THE LOWER 3 BITS OF THE BIT NUMBER
        TAY                         ; IS THE BIT IN THE BYTE, SAVE IN Y
        TXA                         ;RESTORE BIT NUMBER
        LSR     A                   ;DIVIDE BY 8 TO DETERMINE BYTE
        LSR     A
        LSR     A
        TAX                         ;SAVE BYTE NUMBER (0 OR 1) IN X

        ;SET THE BIT
        LDA     VALUE,X             ;GET THE BYTE
        ORA     BITMSK,Y            ;SET THE BIT
        STA     VALUE,X

        ;RETURN THE RESULT IN REGISTERS A AND Y

        LDA     VALUE+1
        LDY     VALUE
        RTS
```

```
BITMSK: .BYTE    00000001B          ;BIT 0 = 1
        .BYTE    00000010B          ;BIT 1 = 1
        .BYTE    00000100B          ;BIT 2 = 1
        .BYTE    00001000B          ;BIT 3 = 1
        .BYTE    00010000B          ;BIT 4 = 1
        .BYTE    00100000B          ;BIT 5 = 1
        .BYTE    01000000B          ;BIT 6 = 1
        .BYTE    10000000B          ;BIT 7 = 1


;DATA
VALUE:  .BLOCK   2                  ;TEMPORARY FOR THE DATA WORD


;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION                                        ;
;                                                               ;
;                                                               ;


SC0701:
        LDA      VAL+1              ;LOAD DATA WORD INTO A,Y
        LDY      VAL
        LDX      BITN               ;GET BIT NUMBER IN X
        JSR      BITSET             ;SET THE BIT
        BRK                         ;RESULT OF VAL = 5555H AND BITN = 0F
                                    ; REGISTER A = D5H, REGISTER Y = 55H
        JMP      SC0701

;TEST DATA, CHANGE FOR DIFFERENT VALUES
VAL:    .WORD    5555H
BITN:   .BYTE    0FH

        .END     ;PROGRAM
```

Bit Clear (BITCLR)

Clears a specified bit in a 16-bit word.

Procedure: the program uses bits 0 through 2 of register X to determine which bit position to clear and bit 3 to select a particular byte of the original word-length data. It then logically ANDs the selected byte with a mask containing a 0 in the chosen bit position and 1s elsewhere. The masks with one 0 bit are available in a table.

Registers Used: All

Execution Time: 57 cycles

Program Size: 42 bytes

Data Memory Required: Two bytes anywhere in RAM (starting at address VALUE).

Special Case: Bit positions above 15 will be interpreted mod 16. That is, for example, bit position 16 is equivalent to bit position 0.

Entry Conditions

More significant byte of data in accumulator
Less significant byte of data in register Y
Bit number to clear in register X

Exit Conditions

More significant byte of result in accumulator
Less significant byte of result in register Y

Examples

1. Data: (A) = $6E_{16}$ = 01101110_2 (more significant byte)
   (Y) = 39_{16} = 00111001_{16} (less significant byte)
   (X) = $0E_{16}$ = 14_{10} (bit position to clear)

   Result: (A) = $2E_{16}$ = 01101110_2 (more significant byte, bit 14 cleared)
   (Y) = 39_{16} = 00111001_2 (less significant byte)

2. Data: (A) = $6E_{16}$ = 01101110_{16} (more significant byte)
   (Y) = 39_{16} = 00111001_2 (less significant byte)
   (X) = 04_{16} = 4_{10} (bit position to clear)

   Result: (A) = $6E_{16}$ = 01101110_2 (more significant byte)
   (Y) = 29_{16} = 00101001_2 (less significant byte, bit 4 cleared)

```
;       Title               Bit clear                                  ;
;       Name:               BITCLR                                     ;
;                                                                      ;
;                                                                      ;

;                                                                      ;
;       Purpose:            Clear a bit in a 16 bit word.              ;
;                                                                      ;
;       Entry:              Register A = High byte of word             ;
;                           Register Y = Low byte of word              ;
;                           Register X = Bit number to clear           ;
;                                                                      ;
;       Exit:               Register A = High byte of word with bit cleared ;
;                           Register Y = Low byte of word with bit cleared  ;
;                                                                      ;
;       Registers used: All                                            ;
;                                                                      ;
;       Time:               57 cycles                                  ;
;                                                                      ;
;       Size:               Program 42 bytes                           ;
;                           Data     2 bytes                           ;
;                                                                      ;
;                                                                      ;


BITCLR:
        ;SAVE THE DATA WORD
        STA     VALUE+1
        STY     VALUE

        ;BE SURE THAT THE BIT NUMBER IS BETWEEN 0 AND 15
        TXA
        AND     #0FH

        ;DETERMINE WHICH BYTE AND WHICH BIT IN THAT BYTE
        TAX                     ;SAVE BIT NUMBER IN X
        AND     #07H            ;THE LOWER 3 BITS OF THE BIT NUMBER
        TAY                     ; IS THE BIT IN THE BYTE, SAVE IN Y
        TXA                     ;RESTORE BIT NUMBER
        LSR     A               ;DIVIDE BY 8 TO DETERMINE BYTE
        LSR     A
        LSR     A
        TAX                     ;SAVE BYTE NUMBER (0 OR 1) IN X

        ;CLEAR THE BIT
        LDA     VALUE,X         ;GET THE BYTE
        AND     BITMSK,Y        ;CLEAR THE BIT
        STA     VALUE,X

        ;RETURN THE RESULT IN REGISTERS A AND Y

        LDA     VALUE+1
        LDY     VALUE
        RTS
```

```
BITMSK: .BYTE    11111110B          ;BIT 0 = 0
        .BYTE    11111101B          ;BIT 1 = 0
        .BYTE    11111011B          ;BIT 2 = 0
        .BYTE    11110111B          ;BIT 3 = 0
        .BYTE    11101111B          ;BIT 4 = 0
        .BYTE    11011111B          ;BIT 5 = 0
        .BYTE    10111111B          ;BIT 6 = 0
        .BYTE    01111111B          ;BIT 7 = 0

;DATA
VALUE:  .BLOCK   2                  ;TEMPORARY FOR THE DATA WORD

;                                                                     ;
;                                                                     ;
;       SAMPLE EXECUTION                                              ;
;                                                                     ;
;                                                                     ;


SC0702:
        LDA      VAL+1              ;LOAD DATA WORD INTO A,Y
        LDY      VAL
        LDX      BITN               ;GET BIT NUMBER IN X
        JSR      BITCLR             ;CLEAR THE BIT
        BRK                         ;RESULT OF VAL = 5555H AND BITN = 00H IS
                                    ; REGISTER A = 55H, REGISTER Y = 54H
        JMP      SC0702

;TEST DATA, CHANGE FOR DIFFERENT VALUES
VAL:    .WORD    5555H
BITN:   .BYTE    0

        .END     ;PROGRAM
```

Sets the Carry flag to the value of a specified bit in a 16-bit word.

Procedure: The program uses bits 0 through 2 of register X to determine which bit position to test and bit 3 to select a particular byte of the original word-length data. It then logically ANDs the selected byte with a mask containing a 1 in the chosen bit position and 0s elsewhere. Since the result is zero if the tested bit is 0 and non-zero if the tested bit is 1, the Zero flag is set to the complement of the tested bit. Finally, the program sets the Carry flag to the complement of the Zero flag, thus making it the same as the tested bit through a double inversion.

Registers Used: All

Execution Time: Approximately 50 cycles

Program Size: 37 bytes

Data Memory Required: Two bytes anywhere in RAM (starting at address VALUE).

Special Case: Bit positions above 15 will be interpreted mod 16. That is, for example, bit position 16 is equivalent to bit position 0.

Entry Conditions

More significant byte of data in accumulator

Less significant byte of data in register Y

Bit position to test in register X

Exit Conditions

Carry set to value of specified bit position in data.

Examples

1. Data: $(A) = 6E_{16} = 01101110_2$ (more significant byte)
   $(Y) = 39_{16} = 00111001_2$ (less significant byte)
   $(X) = 0B_{16} = 11_{10}$ (bit position to test)

   Result: Carry = 1 (value of bit 11)

2. Data: $(A) = 6E_{16} = 01101110_2$ (more significant byte)
   $(Y) = 39_{16} = 00111001_2$ (less significant byte)
   $(X) = 06_{16} = 6_{10}$ (bit position to test)

   Result: Carry = 0 (value of bit 6)

```
;       Title               Bit test                                      ;
;       Name:               BITTST                                        ;
;                                                                         ;
;                                                                         ;

;                                                                         ;
;       Purpose:            Test a bit in a 16 bit word.                  ;
;                                                                         ;
;       Entry:              Register A = High byte of word                ;
;                           Register Y = Low byte of word                 ;
;                           Register X = Bit number to test               ;
;                                                                         ;
;       Exit:               CARRY = value of the tested bit               ;
;                                                                         ;
;       Registers used: All                                               ;
;                                                                         ;
;       Time:               Approximately 50 cycles                       ;
;                                                                         ;
;       Size:               Program 37 bytes                              ;
;                           Data     2 bytes                              ;
;                                                                         ;
;                                                                         ;

BITTST:
        ;SAVE THE DATA WORD
        STA     VALUE+1
        STY     VALUE

        ;BE SURE THAT THE BIT NUMBER IS BETWEEN 0 AND 15
        TXA
        AND     #0FH

        ;DETERMINE WHICH BYTE AND WHICH BIT IN THAT BYTE
        TAX                         ;SAVE BIT NUMBER IN X
        AND     #07H                ;THE LOWER 3 BITS OF THE BIT NUMBER
        TAY                         ; IS THE BIT IN THE BYTE, SAVE IN Y
        TXA                         ;RESTORE BIT NUMBER
        LSR     A                   ;DIVIDE BY 8 TO DETERMINE BYTE
        LSR     A
        LSR     A
        TAX                         ;SAVE BYTE NUMBER (0 OR 1) IN X

        ;SET THE ZERO FLAG TO THE COMPLEMENT OF THE BIT
        LDA     VALUE,X             ;GET THE BYTE
        AND     BITMSK,Y            ;GET THE BIT
                                    ;IF THE BIT IS 0 REGISTER A IS 0 AND Z IS 1
                                    ;ELSE REGISTER A IS NOT 0 AND Z IS 0

        ;SET THE CARRY FLAG TO THE COMPLEMENT OF THE ZERO FLAG

        CLC                         ;ASSUME THE BIT IS 0
        BNE     EXIT                ;BRANCH IF THE BIT IS 0
        SEC                         ;ELSE THE BIT WAS 1
EXIT:
        RTS
```

```
BITMSK: .BYTE    00000001B          ;BIT 0 = 1
        .BYTE    00000010B          ;BIT 1 = 1
        .BYTE    00000100B          ;BIT 2 = 1
        .BYTE    00001000B          ;BIT 3 = 1
        .BYTE    00010000B          ;BIT 4 = 1
        .BYTE    00100000B          ;BIT 5 = 1
        .BYTE    01000000B          ;BIT 6 = 1
        .BYTE    10000000B          ;BIT 7 = 1

;DATA
VALUE:  .BLOCK   2                  ;TEMPORARY FOR THE DATA WORD

;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION                                             ;
;                                                                    ;
;                                                                    ;

SC0703:
        LDA      VAL+1              ;LOAD DATA WORD INTO A,Y
        LDY      VAL
        LDX      BITN               ;GET BIT NUMBER IN X
        JSR      BITTST             ;TEST THE BIT
        BRK                         ;RESULT OF VAL = 5555H AND BITN = 01 IS
                                    ;CARRY = 0
        JMP      SC0703

;TEST DATA, CHANGE FOR DIFFERENT VALUES
VAL:    .WORD    5555H
BITN:   .BYTE    01H

        .END     ;PROGRAM
```

Bit Field Extraction (BFE)

Extracts a field of bits from a word and returns the field in the least significant bit positions. The width of the field and its starting bit position are specified.

Procedure: The program obtains a mask with the specified number of 1 bits from a table, shifts the mask left to align it with the specified starting bit position, and obtains the field by logically ANDing the mask and the data. It then normalizes the bit field by shifting it right so that it starts in bit 0.

Registers Used: All

Execution Time: 34 * STARTING BIT POSITION plus 138 cycles overhead. The starting bit position determines the number of times the mask must be shifted left and the bit field right. For example, if the field starts in bit 6, the execution time is

$$34 * 6 + 138 = 204 + 138 = 342 \text{ cycles}$$

Program Size: 134 bytes

Data Memory Required: Six bytes anywhere in RAM for the index (one byte at address INDEX), the width of the field (one byte at address WIDTH), the data value (two bytes starting at address VALUE), and the mask (two bytes starting at address MASK).

Special Cases:

1. Requesting a field that would extend beyond the end of the word causes the program to return with only the bits through bit 15. That is, no wraparound is provided. If, for example, the user asks for a 10-bit field starting at bit 8, the program will return only 8 bits (bits 8 through 15).

2. Both the starting bit position and the number of bits in the field are interpreted mod 16. That is, for example, bit position 17 is equivalent to bit position 1 and a field of 20 bits is equivalent to a field of 4 bits.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Starting (lowest) bit position of field

Number of bits in the field

Less significant byte of data value
More significant byte of data value

Exit Conditions

More significant byte of bit field in accumulator

Less significant byte of bit field in register Y

Examples

1. Data: Value = $F67C_{16}$ = 1111011001111100_2
Starting bit position = 4
Number of bits in the field = 8

Result: Bit field = 0067_{16} = 0000000001100111_2
We have extracted 8 bits from the original data, starting with bit 4 (that is, bits 4 through 11).

2. Data: Value = $A2D4_{16} = 1010001011010100_2$
Starting bit position = 6
Number of bits in the field = 5

Result: Bit field = $000B_{16} = 0000000000001011_2$
We have extracted 5 bits from the original data, starting with bit 6 (that is, bits 6 through 10).

```
;       Title           Bit Field Extraction                            ;
;       Name:           BFE                                             ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Extract a field of bits from a 16 bit word and  ;
;                       return the field normalized to bit 0.           ;
;                       NOTE: IF THE REQUESTED FIELD IS TOO LONG, THEN  ;
;                             ONLY THE BITS THROUGH BIT 15 WILL BE      ;
;                             RETURNED. FOR EXAMPLE IF A 4 BIT FIELD IS ;
;                             REQUESTED STARTING AT BIT 15 THEN ONLY 1  ;
;                             BIT (BIT 15) WILL BE RETURNED.            ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Starting (lowest) bit position in the field   ;
;                           (0..15),                                    ;
;                         Number of bits in the field (1..16),          ;
;                         Low byte of data word,                        ;
;                         High byte of data word,                       ;
;                                                                       ;
;       Exit:           Register A = High byte of field                 ;
;                       Register Y = Low byte of field                  ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           138 cycles overhead plus                        ;
;                          (34 * starting bit position) cycles          ;
;                                                                       ;
;       Size:           Program 134 bytes                               ;
;                       Data      6 bytes                               ;
;                                                                       ;
;                                                                       ;

BFE:

        ;SAVE RETURN ADDRESS IN Y,X
        PLA
        TAY
        PLA
        TAX
```

```
        ;GET THE STARTING BIT POSITION OF THE FIELD
        PLA
        AND     #0FH            ;MAKE SURE INDEX IS A VALUE BETWEEN 0 AND 15
        STA     INDEX           ;SAVE INDEX

        ;GET THE NUMBER OF BITS IN THE FIELD (MAP FROM 1..WIDTH TO 0..WIDTH-1)
        PLA
        SEC
        SBC     #1              ;SUBTRACT 1
        AND     #0FH            ;MAKE SURE IT IS 0 TO 15
        STA     WIDTH           ;SAVE WIDTH

        ;GET THE DATA WORD
        PLA
        STA     VALUE
        PLA
        STA     VALUE+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA

        ;CONSTRUCT THE MASK
        ; INDEX INTO THE MASK ARRAY USING THE WIDTH PARAMETER
        LDA     WIDTH
        ASL     A               ;MULTIPLY BY 2 SINCE MASKS ARE WORD-LENGTH
        TAY
        LDA     MSKARY,Y
        STA     MASK
        INY
        LDA     MSKARY,Y
        STA     MASK+1

        ;SHIFT MASK LEFT INDEX TIMES TO ALIGN IT WITH THE BEGINNING
        ; OF THE FIELD
        LDY     INDEX
        BEQ     GETFLD          ;BRANCH IF INDEX = 0
SHFTLP:
        ASL     MASK            ;SHIFT LOW BYTE, CARRY := BIT 7
        ROL     MASK+1          ;ROTATE HIGH BYTE, BIT 0 := CARRY
        DEY
        BNE     SHFTLP          ;CONTINUE UNTIL INDEX = 0

        ;GET THE FIELD BY ANDING THE MASK AND THE VALUE
GETFLD:
        LDA     VALUE
        AND     MASK            ;AND LOW BYTE OF VALUE WITH MASK
        STA     VALUE           ;STORE IN VALUE
        LDA     VALUE+1
        AND     MASK+1          ;AND HIGH BYTE OF VALUE WITH MASK
        STA     VALUE+1         ;STORE IT
```

```
        ;NORMALIZE THE FIELD TO BIT 0 BY SHIFTING RIGHT INDEX TIMES
        LDY     INDEX
        BEQ     EXIT            ;BRANCH IF INDEX = 0
NORMLP:
        LSR     VALUE+1         ;SHIFT HIGH BYTE RIGHT, CARRY := BIT 0
        ROR     VALUE           ;ROTATE LOW BYTE RIGHT, BIT 7 := CARRY
        DEY
        BNE     NORMLP          ;CONTINUE UNTIL DONE

EXIT:
        LDY     VALUE
        LDA     VALUE+1
        RTS

        ;
        ;MASK ARRAY WHICH IS USED TO CREATE THE MASK
MSKARY:
        .WORD   0000000000000001B
        .WORD   0000000000000011B
        .WORD   0000000000000111B
        .WORD   0000000000001111B
        .WORD   0000000000011111B
        .WORD   0000000000111111B
        .WORD   0000000001111111B
        .WORD   0000000011111111B
        .WORD   0000000111111111B
        .WORD   0000001111111111B
        .WORD   0000011111111111B
        .WORD   0000111111111111B
        .WORD   0001111111111111B
        .WORD   0011111111111111B
        .WORD   0111111111111111B
        .WORD   1111111111111111B

INDEX:  .BLOCK  1               ;INDEX INTO WORD
WIDTH:  .BLOCK  1               ;WIDTH OF FIELD (NUMBER OF BITS)
VALUE:  .BLOCK  2               ;DATA WORD TO EXTRACT THE FIELD FROM
MASK:   .BLOCK  2               ;TEMPORARY FOR CREATING THE MASK


;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;


SC0704:
        LDA     VAL+1
        PHA
        LDA     VAL
        PHA                     ;PUSH THE DATA WORD
        LDA     NBITS
        PHA                     ;PUSH FIELD WIDTH (NUMBER OF BITS)
        LDA     POS
```

```
        PHA                         ;PUSH INDEX TO FIRST BIT OF THE FIELD
        JSR     BFE                 ;EXTRACT
        BRK                         ;RESULT FOR VAL = 1234H, NBITS = 4, POS = 4 IS
                                    ;  REGISTER A = 0, REGISTER Y = 3
        JMP     SC0704

;TEST DATA, CHANGE FOR OTHER VALUES

        VAL:    .WORD   01234H
        NBITS:  .BYTE   4
        POS:    .BYTE   4

                .END    ;PROGRAM
```

Bit Field Insertion (BFI)

Inserts a field of bits into a word. The width of the field and its starting (lowest) bit position are specified.

Procedure: The program obtains a mask with the specified number of 0 bits from a table. It then shifts the mask and the bit field left to align them with the specified starting bit position. It logically ANDs the mask and the original data word, thus clearing the required bit positions, and then logically ORs the result with the shifted bit field.

Registers Used: All

Execution Time: 31 * STARTING BIT POSITION plus 142 cycles overhead. The starting bit position of the field determines how many times the mask and the field must be shifted left. For example, if the field is inserted starting in bit 10, the execution time is

$$31 * 10 + 142 = 310 + 142 = 452 \text{ cycles.}$$

Program Size: 130 bytes

Data Memory Required: Eight bytes anywhere in RAM for the index (one byte at address INDEX), the width of the field (one byte at address WIDTH), the value to be inserted (two bytes starting at address INSVAL), the data value (two bytes starting at address VALUE), and the mask (two bytes starting at address MASK).

Special Cases:

1. Attempting to insert a field that would extend beyond the end of the word causes the program to insert only the bits through bit 15. That is, no wraparound is provided. If, for example, the user attempts to insert a 6-bit field starting at bit 14, only 2 bits (bits 14 and 15) are actually replaced.

2. Both the starting bit position and the length of the bit field are interpreted mod 16. That is, for example, bit position 17 is the same as bit position 1 and a 20-bit field is the same as a 4-bit field.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Lowest bit position (starting position) of field

Number of bits in the field

Less significant byte of bit field (value to insert)
More significant byte of bit field (value to insert)

Less significant byte of original data value
More significant byte of original data value

Exit Conditions

More significant byte of result in accumulator

Less significant byte of result in register Y

The result is the original data value with the bit field inserted, starting at the specified bit position.

Examples

1. Data: Value = $F67C_{16}$ = 1111011001111100_2
   Starting bit position = 4
   Number of bits in the field = 8
   Bit field = $008B_{16}$ = 0000000010001011_2

   Result: Value with bit field inserted = $F8BC_{16}$ = 1111100010111100_2
   The 8-bit field has been inserted into the original value starting at bit 4 (that is, into bits 4 through 11).

2. Data: Value = $A2D4_{16}$ = 1010001011010100_2
   Starting bit position = 6
   Number of bits in the field = 5
   Bit field = 0015_{16} = 0000000000010101_2

   Result: Value with bit field inserted = $A554_{16}$ = 1010010101010100_2
   The 5-bit field has been inserted into the original value starting at bit 6 (that is, into bits 6 through 10). Those five bits were 01011_2 ($0B_{16}$) and are now 10101_2 (15_{16}).

```
;       Title              Bit Field Insertion                      ;
;       Name:              BFI                                      ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Purpose:           Insert a field of bits which is normalized to ;
;                          bit 0 into a 16 bit word.                ;
;                          NOTE: IF THE REQUESTED FIELD IS TOO LONG, THEN ;
;                                ONLY THE BITS THROUGH BIT 15 WILL BE ;
;                                INSERTED. FOR EXAMPLE IF A 4 BIT FIELD IS ;
;                                TO BE INSERTED STARTING AT BIT 15 THEN ;
;                                ONLY THE FIRST BIT WILL BE INSERTED AT ;
;                                BIT 15.                            ;
;                                                                   ;
;       Entry:             TOP OF STACK                             ;
;                            Low byte of return address,            ;
;                            High byte of return address,           ;
;                            Bit position at which inserted field will ;
;                             start (0..15),                        ;
;                            Number of bits in the field (1..16),   ;
;                            Low byte of value to insert,           ;
;                            High byte of value to insert,          ;
;                            Low byte of value,                     ;
;                            High byte of value                     ;
;                                                                   ;
;       Exit:              Register A = High byte of value with field ;
;                                       inserted                    ;
;                          Register Y = Low byte of value with field ;
;                                       inserted                    ;
;                                                                   ;
;       Registers used: All                                         ;
```

```
;                                                                  ;
;       Time:           142 cycles overhead plus                   ;
;                        (31 * starting bit position) cycles       ;
;                                                                  ;
;       Size:           Program 130 bytes                          ;
;                       Data      8 bytes                          ;
;                                                                  ;
;                                                                  ;

BFI:
        ;SAVE RETURN ADDRESS IN Y,X
        PLA
        TAY
        PLA
        TAX

        ;GET THE LOWEST BIT NUMBER OF THE FIELD
        PLA
        AND     #0FH            ;MAKE SURE INDEX IS A VALUE BETWEEN 0 AND 15
        STA     INDEX           ;SAVE INDEX

        ;GET THE NUMBER OF BITS IN THE FIELD (MAP FROM 1..WIDTH TO 0..WIDTH-1)
        PLA
        SEC
        SBC     #1              ;SUBTRACT 1
        AND     #0FH            ;MAKE SURE IT IS 0 TO 15
        STA     WIDTH           ;SAVE WIDTH

        ;GET THE VALUE TO BE INSERTED (BIT FIELD)
        PLA
        STA     INSVAL
        PLA
        STA     INSVAL+1

        ;GET THE DATA WORD
        PLA
        STA     VALUE
        PLA
        STA     VALUE+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA

        ;CONSTRUCT THE MASK
        ; INDEX INTO THE MASK ARRAY USING THE WIDTH PARAMETER
        LDA     WIDTH
        ASL     A               ;MULTIPLY BY 2 SINCE MASKS ARE WORD-LENGTH
        TAY
        LDA     MSKARY,Y
        STA     MASK
        INY
        LDA     MSKARY,Y
        STA     MASK+1
```

```
        ;SHIFT MASK AND BIT FIELD LEFT INDEX TIMES TO ALIGN THEM
        ; WITH THE BEGINING OF THE FIELD
        LDY     INDEX
        BEQ     INSERT          ;BRANCH IF INDEX = 0
SHFTLP:
        SEC                     ;FILL THE MASK WITH ONES
        ROL     MASK            ;ROTATE LOW BYTE SHIFTING A 1 TO BIT 0 AND
                                ; BIT 7 TO CARRY
        ROL     MASK+1          ;ROTATE HIGH BYTE, BIT 0 := CARRY
        ASL     INSVAL          ;SHIFT THE INSERT VALUE SHIFTING IN ZEROS
        ROL     INSVAL+1
        DEY
        BNE     SHFTLP          ;CONTINUE UNTIL INDEX = 0

        ;USE THE MASK TO ZERO THE FIELD AND THEN OR IN THE INSERT VALUE
INSERT:

        LDA     VALUE
        AND     MASK            ;AND LOW BYTE OF VALUE WITH MASK
        ORA     INSVAL
        TAY                     ;REGISTER Y = LOW BYTE OF THE NEW VALUE
        LDA     VALUE+1
        AND     MASK+1          ;AND HIGH BYTE OF VALUE WITH MASK
        ORA     INSVAL+1        ;REGISTER A = HIGH BYTE OF THE NEW VALUE

        ;RETURN
        RTS

        ;MASK ARRAY WHICH IS USED TO CREATE THE MASK
MSKARY:
        .WORD   1111111111111110B
        .WORD   1111111111111100B
        .WORD   1111111111111000B
        .WORD   1111111111110000B
        .WORD   1111111111100000B
        .WORD   1111111111000000B
        .WORD   1111111110000000B
        .WORD   1111111100000000B
        .WORD   1111111000000000B
        .WORD   1111110000000000B
        .WORD   1111100000000000B
        .WORD   1111000000000000B
        .WORD   1110000000000000B
        .WORD   1100000000000000B
        .WORD   1000000000000000B
        .WORD   0000000000000000B


INDEX:  .BLOCK  1               ;INDEX INTO WORD
WIDTH:  .BLOCK  1               ;WIDTH OF FIELD
INSVAL: .BLOCK  2               ;VALUE TO INSERT
VALUE:  .BLOCK  2               ;DATA WORD
MASK:   .BLOCK  2               ;TEMPORARY FOR CREATING THE MASK
```

```
;                                                                              ;
;                                                                              ;
;        SAMPLE EXECUTION:                                                     ;
;                                                                              ;
;                                                                              ;

SC0705:
         LDA     VAL+1             ;PUSH THE DATA WORD
         PHA
         LDA     VAL
         PHA
         LDA     VALINS+1          ;PUSH THE VALUE TO INSERT
         PHA
         LDA     VALINS
         PHA
         LDA     NBITS             ;PUSH THE FIELD WIDTH
         PHA
         LDA     POS               ;PUSH THE STARTING POSITION OF THE FIELD
         PHA
         JSR     BFI               ;INSERT
         BRK                       ;RESULT FOR VAL = 1234H, VALINS = 0EH,
                                   ;            NBITS = 4, POS = 0CH IS
                                   ;  REGISTER A = E2H, REGISTER Y = 34H
         JMP     SC0705

;TEST DATA, CHANGE FOR OTHER VALUES
VAL:     .WORD   01234H
VALINS:  .WORD   0EH
NBITS:   .BYTE   04H
POS:     .BYTE   0CH

         .END    ;PROGRAM
```

Multiple-Precision Arithmetic Shift Right (MPASR) 7F

Shifts a multi-byte operand right arithmetically by a specified number of bit positions. The length of the number (in bytes) is 255 or less. The Carry flag is set to the value of the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program obtains the sign bit from the most significant byte, shifts that bit to the Carry, and then rotates the entire operand right one bit, starting with the most significant byte. It repeats the operation for the specified number of shifts.

Registers Used: All

Execution Time: NUMBER OF SHIFTS * (18 + 18 * LENGTH OF OPERAND IN BYTES) + 85 cycles.

If, for example, NUMBER OF SHIFTS = 6 and LENGTH OF OPERAND IN BYTES = 8, the execution time is

6 * (18 + 18 * 8) + 85 = 6 * 162 + 85 = 1057 cycles

Program Size: 69 bytes

Data Memory Required: Three bytes anywhere in RAM plus two bytes on page 0. The three bytes anywhere in RAM are temporary storage for the number of shifts (one byte at address NBITS) and the length of the operand (one byte at address LENGTH) and the most significant byte of the operand (one byte at address MSB). The two bytes on page 0 hold a pointer to the operand (starting at address PTR,00D0$_{16}$ in the listing).

Special Cases:

1. If the length of the operand is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of shifts is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

Less significant byte of starting address of operand (address of its least significant byte)

More significant byte of starting address of operand (address of its least significant byte)

Exit Conditions

Operand shifted right arithmetically by the specified number of bit positions. The original sign bit is extended to the right. The Carry flag is set according to the last bit shifted from the rightmost bit position (or cleared if either the number of shifts or the length of the operand is zero).

Examples

1. Data: Length of operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 04

   Result: Shifted operand = $F85A4C719FE06741_{16}$. This is the original operand shifted right four bits arithmetically (the four most significant bits thus all take on the value of the original sign bit, which was 1).
   Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of shifts = 03

   Result: Shifted operand = $07ED485A_{16}$. This is the original operand shifted right three bits arithmetically (the three most significant bits thus all take on the value of the original sign bit, which was 0).
   Carry = 0, since the last bit shifted from the rightmost bit position was 0.

```
;       Title               Multiple-precision arithmetic shift right        ;
;       Name:               MPASR                                            ;
;                                                                            ;
;                                                                            ;
;                                                                            ;
;       Purpose:            Arithmetic shift right a multi-byte operand      ;
;                           N bits.                                          ;
;                                                                            ;
;       Entry:              TOP OF STACK                                     ;
;                             Low byte of return address,                    ;
;                             High byte of return address,                   ;
;                             Number of bits to shift,                       ;
;                             Length of the operand in bytes,                ;
;                             Low byte of address of the operand,            ;
;                             High byte of address of the operand            ;
;                                                                            ;
;                             The operand is stored with ARRAY[0] as its     ;
;                             least significant byte and ARRAY[LENGTH-1]     ;
;                             its most significant byte.                     ;
;                                                                            ;
;       Exit:               Operand shifted right with the most significant ;
;                           bit propagated.                                  ;
;                           CARRY := Last bit shifted from least             ;
;                                    significant position.                   ;
;                                                                            ;
;       Registers used: All                                                  ;
;                                                                            ;
;       Time:               85 cycles overhead plus                          ;
;                            ((18 * length) + 18) cycles per shift           ;
;                                                                            ;
;       Size:               Program 69 bytes                                 ;
;                           Data     3 bytes plus                            ;
;                                    2 bytes in page zero                    ;
;                                                                            ;
;                                                                            ;
```

```
        ;EQUATES
        PTR:  .EQU        0D0H               ;PAGE ZERO FOR POINTER TO OPERAND

MPASR:
        ;SAVE RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET NUMBER OF BITS
        PLA
        STA     NBITS
        ;GET LENGTH OF OPERAND
        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF THE OPERAND
        PLA
        STA     PTR
        PLA
        STA     PTR+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA                         ;RESTORE RETURN ADDRESS

        ;INITIALIZE
        CLC                         ;CLEAR CARRY
        LDA     LENGTH
        BEQ     EXIT                ;EXIT IF LENGTH OF OPERAND IS 0
        LDA     NBITS
        BEQ     EXIT                ;EXIT IF NUMBER OF BITS TO SHIFT IS 0
                                    ; WITH CARRY CLEAR

        ;DECREMENT POINTER SO THAT THE LENGTH BYTE MAY BE USED BOTH
        ; AS A COUNTER AND THE INDEX
        LDA     PTR
        BNE     MPASR1
        DEC     PTR+1               ;DECREMENT HIGH BYTE IF A BORROW IS NEEDED
MPASR1: DEC     PTR                 ;ALWAYS DECREMENT LOW BYTE

        ;LOOP ON THE NUMBER OF SHIFTS TO PERFORM
        LDY     LENGTH
        LDA     (PTR),Y             ;GET THE MOST SIGNIFICANT BYTE
        STA     MSB                 ;SAVE IT FOR THE SIGN

ASRLP:
        LDA     MSB                 ;GET THE MOST SIGNIFICANT BYTE
        ASL     A                   ;SHIFT BIT 7 TO CARRY FOR SIGN EXTENSION
        LDY     LENGTH              ;Y = INDEX TO LAST BYTE AND THE COUNTER

        ;SHIFT RIGHT ONE BIT
```

```
LOOP:
        LDA     (PTR),Y         ;GET NEXT BYTE
        ROR     A               ;ROTATE BIT 7 := CARRY, CARRY := BIT 0
        STA     (PTR),Y         ;STORE NEW VALUE
        DEY                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE THROUGH ALL THE BYTES

        ;DECREMENT NUMBER OF SHIFTS
        DEC     NBITS           ;DECREMENT SHIFT COUNTER
        BNE     ASRLP           ;CONTINUE UNTIL DONE

EXIT:
        RTS

;DATA SECTION
NBITS:  .BLOCK  1               ;NUMBER OF BITS TO SHIFT
LENGTH: .BLOCK  1               ;LENGTH OF OPERAND IN BYTES
MSB:    .BLOCK  1               ;MOST SIGNIFICANT BYTE

;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;


SC0706:
        LDA     AYADR+1 ;PUSH STARTING ADDRESS OF OPERAND
        PHA
        LDA     AYADR
        PHA

        LDA     #SZAY   ;PUSH LENGTH OF OPERAND
        PHA

        LDA     SHIFTS  ;PUSH NUMBER OF SHIFTS
        PHA
        JSR     MPASR   ;SHIFT
        BRK             ;RESULT OF SHIFTING AY = EDCBA987654321H, 4 BITS IS
                        ;                   AY = FEDCBA98765432H, C=0
                        ;  IN MEMORY AY   = 032H
                        ;            AY+1 = 054H
                        ;            AY+2 = 076H
                        ;            AY+3 = 098H
                        ;            AY+4 = 0BAH
                        ;            AY+5 = 0DCH
                        ;            AY+6 = 0FEH

        JMP     SC0706

;
;DATA SECTION
SZAY:   .EQU    7       ;LENGTH OF OPERAND
SHIFTS: .BYTE   4       ;NUMBER OF SHIFTS
AYADR:  .WORD   AY      ;STARTING ADDRESS OF OPERAND
AY      .BYTE   21H,43H,65H,87H,0A9H,0CBH,0EDH

        .END    ;PROGRAM
```

Shifts a multi-byte operand left logically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. The Carry flag is set to the value of the last bit shifted out of the leftmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program clears the Carry initially (to fill with a 0 bit) and then rotates the entire operand left one bit, starting with the least significant byte. It repeats the operation for the specified number of shifts.

Registers Used: All

Execution Time: NUMBER OF SHIFTS * (16 + 20 * LENGTH OF OPERAND IN BYTES) + 73 cycles.

If, for example, NUMBER OF SHIFTS = 4 and LENGTH OF OPERAND IN BYTES = 6 (i.e., a 4-bit shift of a byte operand) the execution time is

4 * (6 + 20 * 6) + 73 = 4 * (136) + 73 = 617 cycles.

Data Memory Required: Two bytes anywhere in RAM plus two bytes on page 0. The two bytes anywhere in RAM are temporary storage for the number of shifts (one byte at address NBITS) and the length of the operand in bytes (one byte at address LENGTH). The two bytes on page 0 hold a pointer to the operand (starting at address PTR, $00D0_{16}$ in the listing).

Special Cases:

1. If the length of the operand is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.
2. If the number of shifts is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

Less significant byte of starting address of operand (address of its least significant byte)

More significant byte of starting address of operand (address of its least significant byte)

Exit Conditions

Operand shifted left logically by the specified number of bit positions (the least significant bit positions are filled with zeros). The Carry flag is set according to the last bit shifted from the leftmost bit position (or cleared if either the number of shifts or the length of the operand is zero).

Examples

1. Data: Length of operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 04

   Result: Shifted operand = $5A4C719FE06741E0_{16}$. This is the original operand shifted left four bits logically; the four least significant bits are all cleared.

   Carry = 0, since the last bit shifted from the leftmost bit position was 0.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of shifts = 03

   Result: Shifted operand = $FB521698_{16}$. This is the original operand shifted left three bits logically; the three least significant bits are all cleared.

   Carry = 1, since the last bit shifted from the leftmost bit position was 1.

```
;       Title              Multiple-precision logical shift left          ;
;       Name:              MPLSL                                          ;
;                                                                         ;
;                                                                         ;
;                                                                         ;
;                                                                         ;
;       Purpose:           Logical shift left a multi-byte operand N bits ;
;                                                                         ;
;       Entry:             TOP OF STACK                                   ;
;                            Low byte of return address,                  ;
;                            High byte of return address,                 ;
;                            Number of bits to shift,                     ;
;                            Length of the operand in bytes,              ;
;                            Low byte of address of the operand,          ;
;                            High byte of address of the operand          ;
;                                                                         ;
;                            The operand is stored with ARRAY[0] as its   ;
;                            least significant byte and ARRAY[LENGTH-1]   ;
;                            its most significant byte.                   ;
;                                                                         ;
;       Exit:              Operand shifted left filling the least         ;
;                          significant bits with zeros.                   ;
;                          CARRY := Last most significant bit             ;
;                                                                         ;
;       Registers used: All                                               ;
;                                                                         ;
;       Time:              73 cycles overhead plus                        ;
;                            ((20 * length) + 16) cycles per shift        ;
;                                                                         ;
;       Size:              Program 54 bytes                               ;
;                          Data     2 bytes plus                          ;
;                                   2 bytes in page zero                  ;
;                                                                         ;
```

```
        ;EQUATES
        PTR:  .EQU      0D0H            ;PAGE ZERO FOR POINTER TO OPERAND

MPLSL:
        ;SAVE RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET NUMBER OF BITS
        PLA
        STA     NBITS

        ;GET LENGTH OF OPERAND
        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF THE OPERAND
        PLA
        STA     PTR
        PLA
        STA     PTR+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA                     ;RESTORE RETURN ADDRESS

        ;INITIALIZE
        CLC                     ;CLEAR CARRY
        LDA     LENGTH
        BEQ     EXIT            ;EXIT IF LENGTH OF THE OPERAND IS 0
        LDA     NBITS
        BEQ     EXIT            ;EXIT IF NUMBER OF BITS TO SHIFT IS 0
                                ; WITH CARRY CLEAR

        ;LOOP ON THE NUMBER OF SHIFTS TO PERFORM
LSLLP:
        LDY     #0              ;Y = INDEX TO LOW BYTE OF THE OPERAND
        LDX     LENGTH          ;X = NUMBER OF BYTES
        CLC                     ;CLEAR CARRY TO FILL WITH ZEROS

        ;SHIFT LEFT ONE BIT
LOOP:
        LDA     (PTR),Y         ;GET NEXT BYTE
        ROL     A               ;ROTATE BIT 0 := CARRY, CARRY := BIT 7
        STA     (PTR),Y         ;STORE NEW VALUE
        INY                     ;INCREMENT TO NEXT BYTE
        DEX                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE THROUGH ALL THE BYTES

        ;DECREMENT NUMBER OF SHIFTS
        DEC     NBITS           ;DECREMENT SHIFT COUNTER
        BNE     LSLLP           ;CONTINUE UNTIL DONE
```

```
EXIT:
        RTS

;DATA SECTION
NBITS:   .BLOCK   1                   ;NUMBER OF BITS TO SHIFT
LENGTH:  .BLOCK   1                   ;LENGTH OF OPERAND


;                                                                          ;
;                                                                          ;
;       SAMPLE EXECUTION:                                                  ;
;                                                                          ;
;                                                                          ;


SC0707:
        LDA     AYADR+1 ;PUSH STARTING ADDRESS OF OPERAND
        PHA
        LDA     AYADR
        PHA

        LDA     #SZAY   ;PUSH LENGTH OF OPERAND
        PHA

        LDA     SHIFTS  ;PUSH NUMBER OF SHIFTS
        PHA
        JSR     MPLSL   ;SHIFT
        BRK             ;RESULT OF SHIFTING AY = EDCBA987654321H, 4 BITS IS
                        ;                   AY = DCBA9876543210H, C=0
                        ;  IN MEMORY AY    = 010H
                        ;            AY+1  = 032H
                        ;            AY+2  = 054H
                        ;            AY+3  = 076H
                        ;            AY+4  = 098H
                        ;            AY+5  = 0BAH
                        ;            AY+6  = 0DCH
        JMP     SC0707

;
;DATA SECTION
SZAY:    .EQU     7         ;LENGTH OF OPERAND
SHIFTS:  .BYTE    4         ;NUMBER OF SHIFTS
AYADR:   .WORD    AY        ;STARTING ADDRESS OF OPERAND
AY:      .BYTE    21H,43H,65H,87H,0A9H,0CBH,0EDH

         .END     ;PROGRAM
```

Shifts a multi-byte number right logically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. The Carry flag is set to the value of the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program clears the Carry initially (to fill with a 0 bit) and then rotates the entire operand right one bit, starting with the most significant byte. It repeats the operation for the specified number of shifts.

Registers Used: All

Execution Time: NUMBER OF SHIFTS * (14 + 18 * LENGTH OF OPERAND IN BYTES) + 80 cycles.

If, for example, NUMBER OF SHIFTS = 4 and LENGTH OF OPERAND IN BYTES = 8 (i.e., a 4-bit shift of an 8-byte operand), the execution time is

4 * (14 + 18 * 8) + 80 = 4 * (158) + 80 = 712 cycles.

Program Size: 59 bytes

Data Memory Required: Two bytes anywhere in RAM plus two bytes on page 0. The two bytes anywhere in RAM are temporary storage for the number of shifts (one byte at address NBITS) and the length of the operand in bytes (one byte at address LENGTH). The two bytes on page 0 hold a pointer to the operand (starting at address PTR, $00D0_{16}$ in the listing).

Special Cases:

1. If the length of the operand is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.
2. If the number of shifts is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

Less significant byte of starting address of operand (address of its least significant byte)

More significant byte of starting address of operand (address of its least significant byte)

Exit Conditions

Operand shifted right logically by the specified number of bit positions (the most significant bit positions are filled with zeros). The Carry flag is set according to the last bit shifted from the rightmost bit position (or cleared if either the the number of shifts or the length of the operand is zero).

Examples

1. Data: Length of operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 04

   Result: Shifted operand = $085A4C719FE06741_{16}$.
   This is the original operand shifted right four bits logically; the four most significant bits are all cleared.
   Carry = 1, since the last bit shifted from the rightmost position was 1.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of shifts = 03

   Result: Shifted operand = $07ED485A_{16}$.
   This is the original operand shifted right three bits logically; the three least significant bits are all cleared.
   Carry = 0, since the last bit shifted from the rightmost bit position was 0.

```
;       Title               Multiple-Precision logical shift right      ;
;       Name:               MPLSR                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Purpose:            Logical shift right a multi-byte operand N bits ;
;                                                                       ;
;       Entry:              TOP OF STACK                                ;
;                             Low byte of return address,               ;
;                             High byte of return address,              ;
;                             Number of bits to shift,                  ;
;                             Length of the operand in bytes,           ;
;                             Low byte of address of the operand,       ;
;                             High byte of address of the operand       ;
;                                                                       ;
;                             The operand is stored with ARRAY[0] as its ;
;                             least significant byte and ARRAY[LENGTH-1] ;
;                             its most significant byte.                ;
;                                                                       ;
;       Exit:               Operand shifted right filling the most      ;
;                           significant bits with zeros                 ;
;                           CARRY := Last bit shifted from the least    ;
;                                    significant position               ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:               85 cycles overhead plus                     ;
;                            ((18 * length) + 14) cycles per shift      ;
;                                                                       ;
;       Size:               Program 59 bytes                            ;
;                           Data     2 bytes plus                       ;
;                                    2 bytes in page zero               ;
;                                                                       ;
;                                                                       ;
```

```
        ;EQUATES
        PTR:  .EQU      0D0H            ;PAGE ZERO FOR POINTER TO OPERAND

MPLSR:
        ;SAVE RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET NUMBER OF BITS
        PLA
        STA     NBITS

        ;GET LENGTH OF OPERAND
        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF THE OPERAND
        PLA
        STA     PTR
        PLA
        STA     PTR+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA                             ;RESTORE RETURN ADDRESS

        ;INITIALIZE
        CLC                             ;CLEAR CARRY
        LDA     LENGTH
        BEQ     EXIT                    ;EXIT IF LENGTH OF OPERAND IS 0
        LDA     NBITS
        BEQ     EXIT                    ;EXIT IF NUMBER OF BITS TO SHIFT IS 0
                                        ; WITH CARRY CLEAR

        ;DECREMENT POINTER SO THAT THE LENGTH BYTE MAY BE USED BOTH
        ; AS A COUNTER AND THE INDEX
        LDA     PTR
        BNE     MPLSR1
        DEC     PTR+1                   ;DECREMENT HIGH BYTE IF A BORROW IS NEEDED
MPLSR1: DEC     PTR                     ;ALWAY DECREMENT HIGH BYTE

        ;LOOP ON THE NUMBER OF SHIFTS TO PERFORM
LSRLP:
        LDY     LENGTH                  ;Y = INDEX TO MSB AND COUNTER
        CLC                             ;CLEAR CARRY TO FILL WITH ZEROS

        ;SHIFT RIGHT ONE BIT
LOOP:
        LDA     (PTR),Y                 ;GET NEXT BYTE
        ROR     A                       ;ROTATE BIT 7 := CARRY, CARRY := BIT 0
        STA     (PTR),Y                 ;STORE NEW VALUE
```

```
        DEY                        ;DECREMENT COUNTER
        BNE     LOOP               ;CONTINUE THROUGH ALL THE BYTES

        ;DECREMENT NUMBER OF SHIFTS
        DEC     NBITS              ;DECREMENT SHIFT COUNTER
        BNE     LSRLP              ;CONTINUE UNTIL DONE

EXIT:
        RTS

;DATA SECTION
NBITS:  .BLOCK  1                  ;NUMBER OF BITS TO SHIFT
LENGTH: .BLOCK  1                  ;LENGTH OF OPERAND


;                                                            ;
;                                                            ;
;       SAMPLE EXECUTION:                                    ;
;                                                            ;
;                                                            ;


SC0708:
        LDA     AYADR+1 ;PUSH STARTING ADDRESS OF OPERAND
        PHA
        LDA     AYADR
        PHA

        LDA     #SZAY   ;PUSH LENGTH OF OPERAND
        PHA

        LDA     SHIFTS  ;PUSH NUMBER OF SHIFTS
        PHA
        JSR     MPLSR   ;SHIFT
        BRK             ;RESULT OF SHIFTING AY = EDCBA987654321H, 4 BITS IS
                        ;                   AY = 0EDCBA98765432H, C=0
                        ;  IN MEMORY AY    = 032H
                        ;            AY+1  = 054H
                        ;            AY+2  = 076H
                        ;            AY+3  = 098H
                        ;            AY+4  = 0BAH
                        ;            AY+5  = 0DCH
                        ;            AY+6  = 00EH

        JMP     SC0708

;
;DATA SECTION
SZAY:   .EQU    7       ;LENGTH OF OPERAND
SHIFTS: .BYTE   4       ;NUMBER OF SHIFTS
AYADR:  .WORD   AY      ;STARTING ADDRESS OF OPERAND
AY:     .BYTE   21H,43H,65H,87H,0A9H,0CBH,0EDH

        .END    ;PROGRAM
```

Multiple-Precision Rotate Right (MPRR)

Rotates a multi-byte operand right by a specified number of bit positions (as if the most significant bit and least significant bit were connected directly). The length of the operand in bytes is 255 or less. The Carry flag is set to the value of the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program shifts bit 0 of the least significant byte of the operand to the Carry flag and then rotates the entire operand right one bit, starting with the most significant byte. It repeats the operation for the specified number of shifts.

Registers used: All

Execution Time: NUMBER OF SHIFTS * (21 + 18 * LENGTH OF OPERAND IN BYTES) + 85 cycles.

If for example, NUMBER OF SHIFTS = 6 and LENGTH OF OPERAND IN BYTES = 4 (i.e. a 6-bit shift of a 4-byte operand), the execution time is

6 * (21 + 18 * 4) + 85 = 6 * (93) + 85
+ 643 cycles.

Program Size: 63 bytes

Data Memory Required: Two bytes anywhere in RAM plus two bytes on page 0. The two bytes anywhere in RAM are temporary storage for the number of shifts (one byte at address NBITS) and the length of the operand in bytes (one byte at address LENGTH). The two bytes on page 0 hold a pointer to the operand (starting at address PTR, $00D0_{16}$ in the listing).

Special Cases:

1. If the length of the operand is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of shifts is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

Less significant byte of starting address of operand (address of its least significant byte)

More significant byte of starting address of operand (address of its least significant byte)

Exit Conditions

Operand rotated right by the specified number of bit positions (the most significant bit positions are filled from the least significant bit positions). The Carry flag is set according to the last bit shifted from the rightmost bit position (or cleared if either the number of shifts or the length of the operand is zero).

Examples

1. Data: Length of operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 04

   Result: Shifted operand = $E85A4C719F306741_{16}$. This is the original operand rotated right four bits: the four most significant bits are equivalent to the original four least significant bits.
   Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of shifts = 03

   Result: Shifted operand = $67ED485A_{16}$. This is the original operand rotated right 3 bits; the three most significant bits (011) are equivalent to the original three least significant bits.
   Carry = 0, since the last bit shifted from the rightmost bit position was 0.

```
;       Title               Multiple-precision rotate right           ;
;       Name:               MPRR                                      ;
;                                                                     ;
;                                                                     ;
;                                                                     ;
;       Purpose:            Rotate right a multi-byte operand N bits  ;
;                                                                     ;
;       Entry:              TOP OF STACK                              ;
;                             Low byte of return address,             ;
;                             High byte of return address,            ;
;                             Number of bits to shift,                ;
;                             Length of the operand in bytes,         ;
;                             Low byte of address of the operand,     ;
;                             High byte of address of the operand     ;
;                                                                     ;
;                             The operand is stored with ARRAY[0] as its ;
;                             least significant byte and ARRAY[LENGTH-1] ;
;                             its most significant byte.              ;
;                                                                     ;
;       Exit:               Operand rotated right                     ;
;                           CARRY := Last bit shifted from the least  ;
;                                    significant position             ;
;                                                                     ;
;       Registers used: All                                           ;
;                                                                     ;
;       Time:               85 cycles overhead plus                   ;
;                             ((18 * length) + 21) cycles per shift   ;
;                                                                     ;
;       Size:               Program 63 bytes                          ;
;                           Data     2 bytes plus                     ;
;                                    2 bytes in page zero             ;
;                                                                     ;
;                                                                     ;

;EQUATES
PTR:    .EQU        0D0H            ;PAGE ZERO FOR POINTER TO OPERAND
```

```
MPRR:
        ;SAVE RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET NUMBER OF BITS
        PLA
        STA     NBITS

        ;GET LENGTH OF OPERAND


        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF THE OPERAND
        PLA
        STA     PTR
        PLA
        STA     PTR+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA                     ;RESTORE RETURN ADDRESS

        ;INITIALIZE
        CLC                     ;CLEAR CARRY
        LDA     LENGTH
        BEQ     EXIT            ;EXIT IF LENGTH OF THE OPERAND IS 0
        LDA     NBITS
        BEQ     EXIT            ;EXIT IF NUMBER OF BITS TO SHIFT IS 0
                                ; WITH CARRY CLEAR

        ;DECREMENT POINTER SO THAT THE LENGTH BYTE MAY BE USED BOTH
        ; AS A COUNTER AND THE INDEX
        LDA     PTR
        BNE     MPRR1
        DEC     PTR+1           ;DECREMENT HIGH BYTE IF A BORROW IS NEEDED
MPRR1:  DEC     PTR             ;ALWAYS DECREMENT LOW BYTE

        ;LOOP ON THE NUMBER OF SHIFTS TO PERFORM
RRLP:
        LDY     #1
        LDA     (PTR),Y         ;GET LOW BYTE OF THE OPERAND
        LSR     A               ;CARRY := BIT 0 OF LOW BYTE
        LDY     LENGTH          ;Y = INDEX TO HIGH BYTE AND COUNTER

        ;ROTATE RIGHT ONE BIT
LOOP:
        LDA     (PTR),Y         ;GET NEXT BYTE
        ROR     A               ;ROTATE BIT 7 := CARRY, CARRY := BIT 0
```

```
        STA     (PTR),Y         ;STORE NEW VALUE
        DEY                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE THROUGH ALL THE BYTES

        ;DECREMENT NUMBER OF SHIFTS
        DEC     NBITS           ;DECREMENT SHIFT COUNTER
        BNE     RRLP            ;CONTINUE UNTIL DONE

EXIT:
        RTS

;DATA SECTION
NBITS:  .BLOCK  1               ;NUMBER OF BITS TO SHIFT
LENGTH: .BLOCK  1               ;LENGTH OF OPERAND


;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;


SC0709:
        LDA     AYADR+1 ;PUSH STARTING ADDRESS OF OPERAND
        PHA
        LDA     AYADR
        PHA

        LDA     #SZAY   ;PUSH LENGTH OF OPERAND
        PHA

        LDA     SHIFTS  ;PUSH NUMBER OF SHIFTS
        PHA
        JSR     MPRR    ;ROTATE
        BRK             ;RESULT OF ROTATING AY = EDCBA987654321H 4 BITS IS
                        ;                   AY = 1EDCBA98765432H C=0
                        ;  IN MEMORY AY   = 032H
                        ;            AY+1 = 054H
                        ;            AY+2 = 076H
                        ;            AY+3 = 098H
                        ;            AY+4 = 0BAH
                        ;            AY+5 = 0DCH
                        ;            AY+6 = 01EH
        JMP     SC0709

;
;DATA SECTION
SZAY:   .EQU    7       ;LENGTH OF OPERAND IN BYTES
SHIFTS: .BYTE   4       ;NUMBER OF SHIFTS
AYADR:  .WORD   AY      ;STARTING ADDRESS OF OPERAND
AY:     .BYTE   21H,43H,65H,87H,0A9H,0CBH,0EDH

        .END    ;PROGRAM
```

Rotates a multi-byte operand left by a specified number of bit positions (i.e., as if the most significant bit and least significant bit were connected directly). The length of the operand in bytes is 255 or less. The Carry flag is set to the value of the last bit shifted out of the leftmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program shifts bit 7 of the most significant byte of the operand to the Carry flag. It then rotates the entire operand left one bit, starting with the least significant byte. It repeats the operation for the specified number of shifts.

Registers Used: All

Execution Time: NUMBER OF SHIFTS * (27 + 20 * LENGTH OF OPERAND IN BYTES) + 73 cycles.

If, for example, NUMBER OF SHIFTS = 4 and LENGTH OF OPERAND IN BYTES = 8 (i.e., a 4-bit shift of an 8-byte operand), the execution time is

4 * (27 + 20 * 8) + 73 = 4 * (187) + 73 = 821 cycles.

Program Size: 60 bytes

Data Memory Required: Two bytes anywhere in RAM plus two bytes on page 0. The two bytes anywhere in RAM are temporary storage for the number of shifts (one byte at address NBITS) and the length of the operand in bytes (one byte at address LENGTH). The two bytes on page 0 hold a pointer to the operand (starting at address PTR, $00D0_{16}$ in the listing).

Special Cases:

1. If the length of the operand is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of shifts is zero, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Number of shifts (bit positions)

Length of the operand in bytes

Less significant byte of starting address of operand (address of its least significant byte)

More significant byte of starting address of operand (address of its least significant byte)

Exit Conditions

Operand rotated left by the specified number of bit positions (the least significant bit positions are filled from the most significant bit positions). The Carry flag is set according to the last bit shifted from the leftmost bit position (or cleared if either the number of shifts or the length of the operand is zero).

Examples

1. Data: Length of operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 04

   Result: Shifted operand = $5A4C719FE06741E8_{16}$. This is the original operand rotated left four bits; the four least significant bits are equivalent to the original four most significant bits.

   Carry = 0, since the last bit shifted from the leftmost bit position was 0.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of shifts = 03

   Result: Shifted operand = $FB521699_{16}$. This is the original operand rotated left three bits; the three least significant bits (001) are equivalent to the original three most significant bits.

   Carry = 1, since the last bit shifted from the leftmost bit position was 1.

```
;       Title           Multiple-precision rotate left              ;
;       Name:           MPRL                                        ;
;                                                                   ;
;                                                                   ;

;                                                                   ;
;       Purpose:        Rotate left a multi-byte operand N bits     ;
;                                                                   ;
;       Entry:          TOP OF STACK                                ;
;                         Low byte of return address,               ;
;                         High byte of return address,              ;
;                         Number of bits to shift,                  ;
;                         Length of the operand in bytes,           ;
;                         Low byte of address of the operand,       ;
;                         High byte of address of the operand       ;
;                                                                   ;
;                         The operand is stored with ARRAY[0] as its ;
;                         least significant byte and ARRAY[LENGTH-1] ;
;                         its most significant byte.                ;
;                                                                   ;
;       Exit:           Number rotated left                         ;
;                       CARRY := Last bit shifted from the most     ;
;                                significant position               ;
;                                                                   ;
;       Registers used: All                                         ;
;                                                                   ;
;       Time:           73 cycles overhead plus                     ;
;                        ((20 * length) + 27) cycles per shift      ;
;                                                                   ;
;       Size:           Program 60 bytes                            ;
;                       Data     2 bytes plus                       ;
;                                2 bytes in page zero               ;
;                                                                   ;
;                                                                   ;

;EQUATES
PTR:    .EQU        0D0H            ;PAGE ZERO FOR POINTER TO OPERAND
```

```
MPRL:
        ;SAVE RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET NUMBER OF BITS
        PLA
        STA     NBITS

        ;GET LENGTH OF OPERAND

        PLA
        STA     LENGTH

        ;GET STARTING ADDRESS OF THE OPERAND
        PLA
        STA     PTR
        PLA
        STA     PTR+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA                     ;RESTORE RETURN ADDRESS

        ;INITIALIZE
        CLC                     ;CLEAR CARRY
        LDA     LENGTH
        BEQ     EXIT            ;EXIT IF THE LENGTH OF THE OPERAND IS 0
        LDA     NBITS
        BEQ     EXIT            ;EXIT IF NUMBER OF BITS TO SHIFT IS 0
                                ; WITH CARRY CLEAR

        ;LOOP ON THE NUMBER OF SHIFTS TO PERFORM
RLLP:
        LDY     LENGTH
        DEY
        LDA     (PTR),Y         ;GET HIGH BYTE OF THE OPERAND
        ASL     A               ;CARRY := BIT 7 OF HIGH BYTE
        LDY     #0              ;Y = INDEX TO LEAST SIGNIFICANT BYTE
        LDX     LENGTH          ;X = NUMBER OF BYTES

        ;ROTATE LEFT ONE BIT
LOOP:
        LDA     (PTR),Y         ;GET NEXT BYTE
        ROL     A               ;ROTATE BIT 7 := CARRY, CARRY := BIT 0
        STA     (PTR),Y         ;STORE NEW VALUE
        INY                     ;INCREMENT TO NEXT BYTE
        DEX                     ;DECREMENT COUNTER
        BNE     LOOP            ;CONTINUE THROUGH ALL THE BYTES

        ;DECREMENT NUMBER OF SHIFTS
        DEC     NBITS           ;DECREMENT SHIFT COUNTER
        BNE     RLLP            ;CONTINUE UNTIL DONE
```

```
EXIT:
        RTS

;DATA SECTION
NBITS:  .BLOCK  1                       ;NUMBER OF BITS TO SHIFT
LENGTH: .BLOCK  1                       ;LENGTH OF OPERAND

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC0710:
        LDA     AYADR+1 ;PUSH STARTING ADDRESS OF OPERAND
        PHA
        LDA     AYADR
        PHA

        LDA     #SZAY   ;PUSH LENGTH OF OPERAND
        PHA

        LDA     SHIFTS  ;PUSH NUMBER OF SHIFTS
        PHA
        JSR     MPRL    ;ROTATE
        BRK             ;RESULT OF ROTATING AY = EDCBA987654321H, 4 BITS IS
                        ;                   AY = DCBA987654321EH, C=0
                        ;  IN MEMORY AY   = 01EH
                        ;            AY+1 = 032H
                        ;            AY+2 = 054H
                        ;            AY+3 = 076H
                        ;            AY+4 = 098H
                        ;            AY+5 = 0BAH
                        ;            AY+6 = 0DCH
        JMP     SC0710

;
;DATA SECTION
SZAY:   .EQU    7       ;LENGTH OF OPERAND IN BYTES
SHIFTS: .BYTE   4       ;NUMBER OF SHIFTS
AYADR:  .WORD   AY      ;ADDRESS OF OPERAND
AY:     .BYTE   21H,43H,65H,87H,0A9H,0CBH,0EDH

        .END    ;PROGRAM
```

Compares two strings and sets the Carry and Zero flags appropriately. The Zero flag is set to 1 if the strings are identical and to 0 otherwise. The Carry flag is set to 0 if the string with the address higher in the stack (string 2) is larger than the other string (string 1); the Carry flag is set to 1 otherwise. The strings are a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length. If the two strings are identical through the length of the shorter, then the longer string is considered to be larger.

Procedure: The program first determines which string is shorter from the lengths which precede the actual characters. It then compares the strings one byte at a time through the length of the shorter. If the program finds corresponding bytes that are not the same through the length of the shorter, the program sets the flags by comparing the lengths.

Registers Used: All

Execution Time:

1. If the strings are not identical through the length of the shorter, the approximate execution time is

81 + 19 * NUMBER OF CHARACTERS COMPARED.

If, for example, the routine compares five characters before finding a difference, the execution time is

81 + 19 * 5 = 81 + 95 = 176 cycles.

2. If the strings are identical through the length of the shorter, the approximate execution time is

93 + 19 * LENGTH OF SHORTER STRING.

If, for example, the shorter string is eight bytes long, the execution time is

93 + 19 * 8 = 93 + 152 = 245 cycles.

Program Size: 52 bytes

Data Memory Required: Four bytes on page 0, two bytes starting at address S1ADR ($00D0_{16}$ in the listing) for a pointer to string 1 and two bytes starting at address S2ADR ($00D2_{16}$ in the listing) for a pointer to string 2.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of starting address of string 2

More significant byte of starting address of string 2

Less significant byte of starting address of string 1

More significant byte of starting address of string 1

Exit Conditions

Flags set as if string 2 had been subtracted from string 1 or, if the strings are equal through the length of the shorter, as if the length of string 2 had been subtracted from the length of string 1.

Zero flag = 1 if the strings are identical, 0 if they are not identical.

Carry flag = 0 if string 2 is larger than string 1, 1 if they are identical or string 1 is larger. If the strings are the same through the length of the shorter, the longer one is considered to be larger.

Examples

1. Data: String 1 = 05'PRINT' (05 is the length of the string)
   String 2 = 03'END' (03 is the length of the string)

   Result: Zero flag = 0 (strings are not identical)
   Carry flag = 1 (string 2 is not larger than string 1)

2. Data: String 1 = 05'PRINT' (05 is the length of the string)
   String 2 = 02'PR' (02 is the length of the string)

   Result: Zero flag = 0 (strings are not identical)
   Carry flag = 1 (string 2 is not larger than string 1)

The longer string (string 1) is considered to be larger. If you want to determine whether string 2 is an abbreviation of string 1, you could use Subroutine 8C (FIND THE POSITION OF A SUBSTRING) and determine whether string 2 was part of string 1 and started at the first character.

3. Data: String 1 = 05'PRINT' (05 is the length of the string)
   String 2 = 06'SYSTEM' (06 is the length of the string)

   Result: Zero flag = 0 (strings are not identical)
   Carry flag = 0 (string 2 is larger than string 1)

We are assuming here that the strings consist of ASCII characters. Note that the byte preceding the actual characters contains a hexadecimal number (the length of the string), not a character. We have represented this byte as two hexadecimal digits in front of the string; the string itself is surrounded by single quotation marks.

Note also that this particular routine treats spaces like any other characters. If for example, the strings are ASCII, the routine will find that SPRINGMAID is larger than SPRING MAID, since an ASCII M ($4D_{16}$) is larger than an ASCII space (20_{16}).

```
;       Title           String compare                                  ;
;       Name:           STRCMP                                          ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Purpose:        Compare 2 strings and return C and Z flags set ;
;                       or cleared.                                     ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Low byte of string 2 address,                 ;
;                         High byte of string 2 address,                ;
;                         Low byte of string 1 address,                 ;
;                         High byte of string 1 address                 ;
;                                                                       ;
;                         A string is a maximum of 255 bytes long plus ;
;                         a length byte which precedes it.              ;
;                                                                       ;
;       Exit:           IF string 1 = string 2 THEN                     ;
;                         Z=1,C=1                                       ;
```

```
;                        IF string 1 > string 2 THEN                ;
;                          Z=0,C=1                                  ;
;                        IF string 1 < string 2 THEN                ;
;                          Z=0,C=0                                  ;
;                                                                   ;
;       Registers used: All                                         ;
;                                                                   ;
;       Time:            Worst case timing for strings which are equal.  ;
;                          93 cycles maximum overhead plus (19 * length) ;
;                                                                   ;
;       Size:            Program 52 bytes                           ;
;                        Data     4 bytes in page zero              ;
;                                                                   ;
;                                                                   ;

;EQUATES
S1ADR   .EQU    0D0H            ;PAGE ZERO POINTER TO STRING 1
S2ADR   .EQU    0D2H            ;PAGE ZERO POINTER TO STRING 2


STRCMP:
        ;GET RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET THE STARTING ADDRESS OF STRING 2
        PLA
        STA     S2ADR
        PLA
        STA     S2ADR+1

        ;GET THE STARTING ADDRESS OF STRING 1
        PLA
        STA     S1ADR
        PLA
        STA     S1ADR+1

        ;RESTORE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA

        ;
        ;DETERMINE WHICH STRING IS SHORTER
        LDY     #0
        LDA     (S1ADR),Y       ;GET LENGTH OF STRING #1
        CMP     (S2ADR),Y
        BCC     BEGCMP          ;IF STRING #2 IS SHORTER THEN
        LDA     (S2ADR),Y       ; USE ITS LENGTH INSTEAD

        ;
        ;COMPARE THE STRINGS THROUGH THE LENGTH OF THE SHORTER STRING
```

```
BEGCMP:
        TAX                             ;X IS THE LENGTH OF THE SHORTER STRING
        BEQ     TSTLEN                  ;BRANCH IF LENGTH IS ZERO

        LDY     #1                      ;POINT AT FIRST CHARACTER OF STRINGS
CMPLP:
        LDA     (S1ADR),Y
        CMP     (S2ADR),Y
        BNE     EXIT                    ;BRANCH IF CHARACTERS ARE NOT EQUAL
                                        ;  Z,C WILL BE PROPERLY SET OR CLEARED
                                        ;ELSE
        INY                             ; NEXT CHARACTER
        DEX                             ; DECREMENT COUNTER
        BNE     CMPLP                   ; CONTINUE UNTIL ALL BYTES ARE COMPARED

        ;
        ;THE 2 STRINGS ARE EQUAL TO THE LENGTH OF THE SHORTER
        ;SO USE THE LENGTHS AS THE BASIS FOR SETTING THE FLAGS
TSTLEN:
        LDY     #0                      ;COMPARE LENGTHS
        LDA     (S1ADR),Y
        CMP     (S2ADR),Y               ;SET OR CLEAR THE FLAGS

        ;
        ;EXIT FROM STRING COMPARE
EXIT:
        RTS
;                                                                          ;
;                                                                          ;
;       SAMPLE EXECUTION:                                                  ;
;                                                                          ;
;                                                                          ;

SC0801:
        LDA     SADR1+1                 ;PUSH STARTING ADDRESS OF STRING 1
        PHA
        LDA     SADR1
        PHA
        LDA     SADR2+1                 ;PUSH STARTING ADDRESS OF STRING 2
        PHA
        LDA     SADR2
        PHA
        JSR     STRCMP                  ;COMPARE
        BRK                             ;RESULT OF COMPARING "STRING 1" AND "STRING 2"
                                        ;IS STRING 1 LESS THAN STRING 2 SO
                                        ;   Z=0,C=0
        JMP     SC0801                  ;LOOP FOR ANOTHER TEST

;
;TEST DATA, CHANGE TO TEST OTHER VALUES
SADR1   .WORD   S1
SADR2   .WORD   S2
S1      .BYTE   20H,"STRING 1                    "
S2      .BYTE   20H,"STRING 2                    "

        .END    ;PROGRAM
```

String Concatenation (CONCAT) 8B

Combines (concatenates) two strings, placing the second immediately after the first in memory. If the concatenation would produce a string longer than a specified maximum, the program concatenates only enough of string 2 to give the combined string its maximum length. The Carry flag is cleared if all of string 2 can be concatenated and set to 1 if part of string 2 must be dropped. Both strings are a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length.

Procedure: The program uses the length of string 1 to determine where to start adding characters and the length of string 2 to determine how many characters to add. If the sum of the lengths exceeds the maximum, the program indicates an overflow and reduces the number of characters it must add (the number is the maximum length minus the length of string 1). It then moves the appropriate number of characters from string 2 to the end of string 1, updates the length of string 1, and sets the Carry flag to indicate whether any characters had to be discarded.

Registers Used: All

Execution Time: Approximately 40 * NUMBER OF CHARACTERS CONCATENATED plus 164 cycles overhead. The NUMBER OF CHARACTERS CONCATENATED is normally the length of string 2, but will be the maximum length of string 1 minus its current length if the combined string would be longer than the maximum. If, for example, NUMBER OF CHARACTERS CONCATENATED is 14_{16} (20_{10}), the execution time is

$40 * 20 + 161 = 800 + 164 = 964$ cycles.

Program Size: 141 bytes

Data Memory Required: Seven bytes anywhere in RAM plus four bytes on page 0. The seven bytes anywhere in RAM are temporary storage for the maximum length of string 1 (1 byte at address MAXLEN), the length of string 1 (1 byte at address S1LEN), the length of string 2 (1 byte at address S2LEN), a running index for string 1 (1 byte at address S1IDX), a running index for string 2 (1 byte at address S2IDX), a concatenation counter (1 byte at address COUNT), and a flag that indicates whether the combined strings overflowed (1 byte at address STRGOV). The four bytes on page 0 hold pointers to string 1 (two bytes starting at address S1ADR, address $00D0_{16}$ in the listing) and to string 2 (two bytes starting at address S1ADR, address $00D0_{16}$ in the listing).

Special Cases:

1. If the concatenation would result in a string longer than the specified maximum length, the program concatenates only enough of string 2 to reach the maximum. If any of string 2 must be truncated, the Carry flag is set to 1.
2. If string 2 has a length of zero, the program exits with the Carry flag cleared (no errors) and string 1 unchanged. That is, a length of zero for either string is interpreted as zero, not 256.
3. If the original length of string 1 exceeds the specified maximum length, the program exits with the Carry flag set to 1 (indicating an error) and string 1 unchanged.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Maximum length of string 1

Less significant byte of starting address of string 2
More significant byte of starting address of string 2

Less significant byte of starting address of string 1
More significant byte of starting address of string 1

Exit Conditions

String 2 concatenated at the end of string 1 and the length of string 1 increased appropriately. If the resulting string would exceed the maximum length, only the part of string 2 that would give string 1 its maximum length is concatenated. If any part of string 2 must be dropped, the Carry flag is set to 1. Otherwise, the Carry flag is cleared.

Examples

1. Data: Maximum length of string 1 = $0E_{16} = 14_{10}$
   String 1 = 07'JOHNSON' (07 is the length of the string)
   String 2 = 05', DON' (05 is the length of the string)

   Result: String 1 = 0C'JOHNSON, DON' ($0C_{16} = 12_{10}$ is the length of the combined string with string 2 placed after string 1).
   Carry = 0, since the concatenation did not produce a string exceeding the maximum length.

2. Data: Maximum length of string 1 = $0E_{16} = 14_{10}$
   String 1 = 07'JOHNSON' (07 is the length of the string)
   String 2 = 09', RICHARD' (09 is the length of the string)

   Result: String 1 = 0E'JOHNSON, RICHA' ($0E_{16} = 14_{10}$ is the maximum length allowed, so the last two characters of string 2 have been dropped.)
   Carry = 1, since the concatenation produced a string longer than the maximum length.

Note that we are representing the initial byte (containing the length of the string) as two hexadecimal digits in both examples.

```
;        Title               String Concatenation                         ;
;        Name:               CONCAT                                       ;
;                                                                         ;
;                                                                         ;

;                                                                         ;
;        Purpose:            Concatenate 2 strings into one string.       ;
;                                                                         ;
;        Entry:              TOP OF STACK                                 ;
;                              Low byte of return address,                ;
;                              High byte of return address,               ;
;                              Maximum length of string 1,                ;
;                              Low byte of string 2 address,              ;
;                              High byte of string 2 address,             ;
;                              Low byte of string 1 address,              ;
;                              High byte of string 1 address              ;
;                                                                         ;
;                              A string is a maximum of 255 bytes long plus ;
;                              a length byte which precedes it.           ;
;                                                                         ;
;        Exit:               string 1 := string 1 concatenated with string 2 ;
;                            If no errors then                            ;
;                              CARRY := 0                                 ;
;                            else                                         ;
;                              begin                                      ;
;                                CARRY := 1                               ;
;                                if the concatenation makes string 1 too  ;
;                                long concatenate only the part of string 2 ;
;                                which will result in string 1 having its ;
;                                maximum length                           ;
;                                if length(string1) > maximum length then ;
;                                  no concatenation is done               ;
;                              end;                                       ;
;                                                                         ;
;        Registers used: All                                              ;
;                                                                         ;
;        Time:               Approximately 40 * (length of string 2) cycles ;
;                            plus 161 cycles overhead                     ;
;                                                                         ;
;        Size:               Program 141 bytes                            ;
;                            Data      7 bytes plus                       ;
;                                      4 bytes in page zero               ;
;                                                                         ;
;                                                                         ;

;EQUATES
S1ADR   .EQU    0D0H            ;PAGE ZERO POINTER TO STRING 1
S2ADR   .EQU    0D2H            ;PAGE ZERO POINTER TO STRING 2

CONCAT:

        ;GET RETURN ADDRESS
        PLA
        TAY                     ;SAVE LOW BYTE
        PLA
        TAX                     ;SAVE HIGH BYTE
```

```
        ;GET MAXIMUM LENGTH OF STRING 1
        PLA
        STA     MAXLEN

        ;GET THE STARTING ADDRESS OF STRING 2
        PLA
        STA     S2ADR
        PLA
        STA     S2ADR+1

        ;GET THE STARTING ADDRESS OF STRING 1
        PLA
        STA     S1ADR
        PLA
        STA     S1ADR+1

        ;RESTORE RETURN ADDRESS
        TXA
        PHA                     ;RESTORE HIGH BYTE
        TYA
        PHA                     ;RESTORE LOW BYTE

        ;DETERMINE WHERE TO START CONCATENATING
        LDY     #0
        LDA     (S1ADR),Y       ;GET CURRENT LENGTH OF STRING 1
        STA     S1LEN
        STA     S1IDX
        INC     S1IDX           ;START CONCATENATING AT THE END OF STRING 1
        LDA     (S2ADR),Y       ;GET LENGTH OF STRING 2
        STA     S2LEN
        LDA     #1
        STA     S2IDX           ;START CONCATENATION AT BEGINNING OF STRING 2

        ;
        ;DETERMINE THE NUMBER OF CHARACTERS TO CONCATENATE
        LDA     S2LEN           ;GET LENGTH OF STRING 2
        CLC
        ADC     S1LEN           ;ADD TO CURRENT LENGTH OF STRING 1
        BCS     TOOLNG          ;BRANCH IF LENGTH WILL EXCEED 255 BYTES
        CMP     MAXLEN          ;CHECK AGAINST MAXIMUM LENGTH
        BEQ     LENOK           ;BRANCH IF LENGTH DOES NOT EXCEED MAXIMUM
        BCC     LENOK

        ;
        ;RESULTING STRING WILL BE TOO LONG SO
        ; INDICATE A STRING OVERFLOW, STRGOV := 0FFH
        ; SET NUMBER OF CHARACTERS TO CONCATENATE =  MAXLEN - S1LEN
        ; SET LENGTH OF STRING 1 TO MAXIMUM LENGTH
TOOLNG:
        LDA     #0FFH
        STA     STRGOV          ;INDICATE OVERFLOW
        LDA     MAXLEN
        SEC
        SBC     S1LEN
        BCC     EXIT            ;EXIT IF MAXIMUM LENGTH < STRING 1 LENGTH
```

```
                                        ; (THE ORIGINAL STRING WAS TOO LONG !!)
        STA     COUNT                   ;SET COUNT TO S1LEN - MAXLEN
        LDA     MAXLEN
        STA     S1LEN                   ;SET LENGTH OF STRING 1 TO MAXIMUM
        JMP     DOCAT                   ;PERFORM CONCATENATION

        ;RESULTING LENGTH DOES NOT EXCEED MAXIMUM
        ; LENGTH OF STRING 1 = S1LEN + S2LEN
        ; INDICATE NO OVERFLOW, STRGOV := 0
        ; SET NUMBER OF CHARACTERS TO CONCATENATE TO LENGTH OF STRING 2
LENOK:
        STA     S1LEN                   ;SAVE THE SUM OF THE 2 LENGTHS
        LDA     #0
        STA     STRGOV                  ;INDICATE NO OVERFLOW
        LDA     S2LEN
        STA     COUNT                   ;COUNT := LENGTH OF STRING 2

        ;
        ;CONCATENATE THE STRINGS
DOCAT:
        LDA     COUNT
        BEQ     EXIT                    ;EXIT IF NO BYTES TO CONCATENATE

CATLP:
        LDY     S2IDX
        LDA     (S2ADR),Y               ;GET NEXT BYTE FROM STRING 2
        LDY     S1IDX
        STA     (S1ADR),Y               ;MOVE IT TO END OF STRING 1
        INC     S1IDX                   ;INCREMENT STRING 1 INDEX
        INC     S2IDX                   ;INCREMENT STRING 2 INDEX
        DEC     COUNT                   ;DECREMENT COUNTER
        BNE     CATLP                   ;CONTINUE UNTIL COUNT = 0

EXIT:
        LDA     S1LEN                   ;UPDATE LENGTH OF STRING 1
        LDY     #0
        STA     (S1ADR),Y
        LDA     STRGOV                  ;GET OVERFLOW INDICATOR
        ROR     A                       ;CARRY = 1 IF OVERLOW, 0 IF NOT
        RTS

;
;DATA
MAXLEN: .BLOCK  1                       ;MAXIMUM LENGTH OF S1
S1LEN:  .BLOCK  1                       ;LENGTH OF S1
S2LEN:  .BLOCK  1                       ;LENGTH OF S2
S1IDX:  .BLOCK  1                       ;RUNNING INDEX INTO S1
S2IDX:  .BLOCK  1                       ;RUNNING INDEX INTO S2
COUNT:  .BLOCK  1                       ;CONCATENATION COUNTER
STRGOV: .BLOCK  1                       ;STRING OVERFLOW FLAG

;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
                                                                        ;
```

```
;                                                                  ;

SC0802:
        LDA     SADR1+1 ;PUSH ADDRESS OF STRING 1
        PHA
        LDA     SADR1
        PHA
        LDA     SADR2+1 ;PUSH ADDRESS OF STRING 2
        PHA
        LDA     SADR2
        PHA
        LDA     #20H    ;PUSH MAXIMUM LENGTH OF STRING 1
        PHA
        JSR     CONCAT  ;CONCATENATE
        BRK             ;RESULT OF CONCATENATING "LASTNAME" AND ", FIRSTNAME"
                        ; IS S1 = 13H,"LASTNAME, FIRSTNAME"
        JMP     SC0802  ;LOOP FOR ANOTHER TEST

;
;TEST DATA, CHANGE FOR OTHER VALUES
SADR1   .WORD   S1              ;STARTING ADDRESS OF STRING 1
SADR2   .WORD   S2              ;STARTING ADDRESS OF STRING 2
S1      .BYTE   8H              ;LENGTH OF S1
        .BYTE   "LASTNAME                        " ;32 BYTE MAX LENGTH
S2      .BYTE   0BH             ;LENGTH OF S2
        .BYTE   ", FIRSTNAME                     " ;32 BYTE MAX LENGTH

        .END    ;PROGRAM
```

Find the Position of a Substring (POS) 8C

Searches for the first occurrence of a substring within a string. Returns the index at which the substring starts if it is found and 0 if it is not found. The string and the substring are both a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length. Thus, if the substring is found, its starting index cannot be less than 1 or more than 255.

Procedure: The program moves through the string searching for the substring until it either finds a match or the remaining part of the string is shorter than the substring and hence cannot possibly contain it. If the substring does not appear in the string, the program clears the accumulator; otherwise, the program places the starting index of the substring in the accumulator.

Registers Used: All

Execution Time: Data-dependent, but the overhead is 135 cycles, each successful match of one character takes 47 cycles, and each unsuccessful match of one character takes 50 cycles. The worst case occurs when the string and substring always match except for the last character in the substring, such as

String = 'AAAAAAAAB'
Substring = 'AAB'

The execution time in that case is

(STRING LENGTH − SUBSTRING LENGTH + 1) * (47 * (SUBSTRING LENGTH − 1) + 50) + 135

If, for example, STRING LENGTH = 9 and SUBSTRING LENGTH = 3, the execution time is

(9 − 3 + 1) * (47 * (3 − 1) + 50) + 135 = 7 * 144 + 135 = 1008 + 135 = 1143 cycles.

Program Size: 124 bytes

Data Memory Required: Six bytes anywhere in RAM plus four bytes on page 0. The six bytes anywhere in RAM are temporary storage for the length of the string (one byte at address SLEN), the length of the substring (one byte at address SUBLEN), a running index into the string (one byte at address SIDX), a running index into the substring (one byte at address SUBIDX), a search counter (one byte at address COUNT), and an index into the string (one byte at address INDEX). The four bytes on page 0 hold pointers to the substring (two bytes starting at address SUBSTG, $00D0_{16}$ in the listing) and to the string (two bytes starting at address STRING, $00D2_{16}$ in the listing).

Special Cases:

1. If either the string or the substring has a length of zero, the program exits with zero in the accumulator, indicating that it did not find the substring.
2. If the substring is longer than the string, the program exits with zero in the accumulator, indicating that it did not find the substring.
3. If the program returns an index of 1, the substring may be regarded as an abbreviation of the string. That is, the substring occurs in the string, starting at the first character. A typical example would be a string PRINT and a substring PR.
4. If the substring occurs more than once in the string, the program will return only the index to the first occurrence (the occurrence with the lowest starting index).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of starting address of substring
More significant byte of starting address of substring

Less significant byte of starting address of string
More significant byte of starting address of string

Exit Conditions

Accumulator contains index at which first occurrence of substring starts if it is found; accumulator contains zero if substring is not found.

Examples

1. Data: String = 1D' ENTER SPEED IN MILES PER HOUR' ($1D_{16} = 29_{10}$ is the length of the string).
   Substring = 05'MILES' (05 is the length of the substring)

   Result: Accumulator contains 10_{16} (16_{10}), the index at which the substring 'MILES' starts.

2. Data: String = 1B'SALES FIGURES FOR JUNE 1981' ($1B_{16} = 27_{10}$ is the length of the string)
   Substring = 04'JUNE' (04 is the length of the substring)

   Result: Accumulator contains 13_{16} (19_{10}), the index at which the substring 'JUNE' starts.

3. Data: String = 10'LET Y1 = X1 + R7' ($10_{16} = 16_{10}$ is the length of the string)
   Substring = 02'R4' (02 is the length of the substring)

   Result: Accumulator contains 00, since the substring 'R4' does not appear in the string LET Y1 = X1 + R7.

4. Data: String =07'RESTORE' (07 is the length of the string)
   Substring = 03'RES' (03 is the length of the substring)

   Result: Accumulator contains 01, the index at which the substring "RES" starts. An index of 01 indicates that the substring could be an abbreviation of the string; such abbreviations are, for example, often used in interactive programs (such as BASIC interpreters) to save on typing and storage.

```
;       Title                  Find the position of a substring in a string   ;
;       Name:                  POS                                            ;
;                                                                             ;
;                                                                             ;
;                                                                             ;
;       Purpose:               Search for the first occurrence of a substring ;
;                              within a string and return its starting index. ;
;                              If the substring is not found a 0 is returned. ;
;                                                                             ;
;       Entry:                 TOP OF STACK                                   ;
;                                Low byte of return address,                  ;
;                                High byte of return address,                 ;
;                                Low byte of substring address,               ;
;                                High byte of substring address,              ;
;                                Low byte of string address,                  ;
;                                High byte of string address                  ;
;                                                                             ;
;                                A string is a maximum of 255 bytes long plus ;
;                                a length byte which precedes it.             ;
;                                                                             ;
;       Exit:                  If the substring is found then                 ;
;                                Register A = its starting index              ;
;                              else                                           ;
;                                Register A = 0                               ;
;                                                                             ;
;       Registers used: All                                                   ;
;                                                                             ;
;                                                                             ;
;       Time:                  Since the algorithm is so data dependent       ;
;                              a simple formula is impossible but the         ;
;                              following statements are true and a            ;
;                              worst case is given below:                     ;
;                                                                             ;
;                              135 cycles overhead.                           ;
;                              Each match of 1 character takes 47 cycles      ;
;                              A mismatch takes 50 cycles.                    ;
;                                                                             ;
;                              Worst case timing will be when the             ;
;                              string and substring always match              ;
;                              except for the last character of the           ;
;                              substring, Such as:                            ;
;                                  string = 'AAAAAAAAAB'                      ;
;                                  substring = 'AAB'                          ;
;                              135 cycles overhead plus                       ;
;                          (length(string) - length(substring) + 1) *         ;
;                                  (((length(substring)-1) * 47) + 50)        ;
;                                                                             ;
;       Size:                  Program 124 bytes                              ;
;                              Data      6 bytes plus                         ;
;                                        4 bytes in page zero                 ;
;                                                                             ;
;                                                                             ;

;EQUATES
```

```
SUBSTG  .EQU    0D0H              ;PAGE ZERO POINTER TO SUBSTRING
STRING  .EQU    0D2H              ;PAGE ZERO POINTER TO STRING

POS:
        ;GET RETURN ADDRESS
        PLA
        TAY                       ;SAVE LOW BYTE
        PLA
        TAX                       ;SAVE HIGH BYTE

        ;GET THE STARTING ADDRESS OF SUBSTRING
        PLA
        STA     SUBSTG
        PLA
        STA     SUBSTG+1

        ;GET THE STARTING ADDRESS OF STRING
        PLA
        STA     STRING
        PLA
        STA     STRING+1

        ;RESTORE RETURN ADDRESS
        TXA
        PHA                       ;RESTORE HIGH BYTE
        TYA
        PHA                       ;RESTORE LOW BYTE

        ;SET UP TEMPORARY LENGTH AND INDEX BYTES
        LDY     #0
        LDA     (STRING),Y        ;GET LENGTH OF STRING
        BEQ     NOTFND            ;EXIT IF LENGTH OF STRING = 0
        STA     SLEN
        LDA     (SUBSTG),Y        ;GET LENGTH OF SUBSTRING
        BEQ     NOTFND            ;EXIT IF LENGTH OF SUBSTRING = 0
        STA     SUBLEN

        ;IF THE SUBSTRING IS LONGER THAN THE STRING DECLARE THE
        ; SUBSTRING NOT FOUND
        LDA     SUBLEN
        CMP     SLEN
        BEQ     LENOK
        BCS     NOTFND            ;CANNOT FIND SUBSTRING IF IT IS LONGER THAN
                                  ; STRING
        ;START SEARCH, CONTINUE  UNTIL REMAINING STRING SHORTER THAN SUBSTRING
LENOK:
        LDA     #1
        STA     INDEX             ;START LOOKING AT FIRST CHARACTER OF STRING
        LDA     SLEN              ;CONTINUE UNTIL REMAINING STRING TOO SHORT
        SEC                       ; COUNT=STRING LENGTH - SUBSTRING LENGTH + 1
        SBC     SUBLEN
        STA     COUNT
        INC     COUNT

        ;SEARCH FOR SUBSTRING IN STRING
SLP1:
```

```
        LDA     INDEX
        STA     SIDX            ;START STRING INDEX AT INDEX
        LDA     #1
        STA     SUBIDX          ;START SUBSTRING INDEX AT 1

        ;LOOK FOR SUBSTRING BEGINNING AT INDEX
CMPLP:
        LDY     SIDX
        LDA     (STRING),Y      ;GET NEXT CHARACTER FROM STRING
        LDY     SUBIDX
        CMP     (SUBSTG),Y      ;COMPARE TO NEXT CHARACTER IN SUBSTRING
        BNE     SLP2            ;BRANCH IF SUBSTRING IS NOT HERE
        LDY     SUBIDX
        CPY     SUBLEN          ;TEST IF WE ARE DONE
        BEQ     FOUND           ;BRANCH IF ALL CHARACTERS WERE EQUAL
        INY                     ;ELSE INCREMENT TO NEXT CHARACTER
        STY     SUBIDX
        INC     SIDX            ;INCREMENT STRING INDEX
        JMP     CMPLP           ;CONTINUE

        ;ARRIVE HERE IF THE SUBSTRING IS NOT YET FOUND
SLP2:
        INC     INDEX           ;INCREMENT INDEX
        DEC     COUNT           ;DECREMENT COUNT
        BNE     SLP1            ;BRANCH IF NOT DONE
        BEQ     NOTFND          ;ELSE EXIT NOT FOUND

FOUND:
        LDA     INDEX           ;SUBSTRING FOUND, A = STARTING INDEX
        JMP     EXIT

NOTFND:
        LDA     #0              ;SUBSTRING NOT FOUND, A = 0

EXIT
        RTS

;
;DATA
SLEN:   .BLOCK  1               ;LENGTH OF STRING
SUBLEN: .BLOCK  1               ;LENGTH OF SUBSTRING
SIDX:   .BLOCK  1               ;RUNNING INDEX INTO STRING
SUBIDX: .BLOCK  1               ;RUNNING INDEX INTO SUBSTRING
COUNT:  .BLOCK  1               ;SEARCH COUNTER
INDEX:  .BLOCK  1               ;CURRENT INDEX INTO STRING

;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION:                                            ;
;                                                                    ;
;                                                                    ;


SC0803:
        LDA     SADR+1          ;PUSH ADDRESS OF THE STRING
```

```
        PHA
        LDA     SADR
        PHA
        LDA     SUBADR+1        ;PUSH ADDRESS OF THE SUBSTRING
        PHA
        LDA     SUBADR
        PHA
        JSR     POS             ;FIND POSITION OF SUBSTRING
        BRK                     ;RESULT OF SEARCHING "AAAAAAAAAB" FOR "AAB" IS
                                ; REGISTER A=8
        JMP     SC0803          ;LOOP FOR ANOTHER TEST

;
;TEST DATA, CHANGE FOR OTHER VALUES
SADR    .WORD   STG
SUBADR  .WORD   SSTG
STG     .BYTE   0AH             ;LENGTH OF STRING
        .BYTE   "AAAAAAAAAB                     " ;32 BYTE MAX LENGTH
SSTG    .BYTE   3H              ;LENGTH OF SUBSTRING
        .BYTE   "AAB                            " ;32 BYTE MAX LENGTH

        .END    ;PROGRAM
```

Copy a Substring from a String (COPY) 8D

Copies a substring from a string, given a starting index and the number of bytes to copy. The strings are a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length. If the starting index of the substring is zero (i.e., the substring would start in the length byte) or is beyond the end of the string, the substring is given a length of zero and the Carry flag is set to 1. If the substring would exceed its maximum length or would extend beyond the end of the string, then only the maximum number or the available number of characters (up to the end of the string) are placed in the substring, and the Carry flag is set tō 1. If the substring can be formed as specified, the Carry flag is cleared.

Procedure: The program exits immediately if the number of bytes to copy, the maximum length of the substring, or the starting index is zero. It also exits immediately if the starting index exceeds the length of the string. If none of these conditions holds, the program checks if the number of bytes to copy exceeds either the maximum length of the substring or the number of characters available in the string. If either one is exceeded, the program reduces the number of bytes to copy appropriately. It then copies the proper number of bytes from the string to the substring. The program clears the Carry flag if the substring can be formed as specified and sets the Carry flag if it cannot.

Registers Used: All

Execution Time: Approximalely 36 * NUMBER OF BYTES COPIES plus 200 cycles overhead. NUMBER OF BYTES COPIED is lhe number specified (if no problems occur) or the number available or the maximum length of the substring if the copying would go beyond the end of either lhe slring or the substring. If, for example, NUMBER OF BYTES COPIED = 12_{10} ($0C_{16}$), the execution time is

36 * 12 + 200 = 432 +200 = 632 cycles.

Program Size: 173 bytes.

Data Memory Required: Six bytes anywhere in RAM plus four bytes on page 0. The six bytes anywhere in RAM hold the lenglh of lhe slring (one byte at address SLEN), the length of the substring (one byle at address DLEN), lhe maximum length of the subslring (one byte at address MAXLEN), the search counter (one byte at address COUNT), the current index into the slring (one byte at address INDEX), and an error flag (one byte at address CPYERR). The four bytes on page 0 hold pointers to the string (two bytes starting at address DSTRG, $00D0_{16}$ in the listing) and lo the substring (two bytes starting at address SSTRG, $00D2_{16}$ in the listing).

Special Cases:

1. If the number of bytes to copy is zero, the program assigns the subslring a length of zero and clears the Carry flag, indicating no error.
2. If the maximum length of the subslring is zero, the program assigns the subslring a length of zero and sels the Carry flag to I, indicaling an error.
3. If lhe starting index of the substring is zero, the program assigns lhe substring a length of zero and sets the Carry flag to I, indicating an error.
4. If the source string does not even reach the specified starting index, the program assigns the substring a length of zero and sets the Carry flag to I, indicating an error.
5. If the substring would extend beyond the end of the source string, the program places all the available characters in the substring and sets lhe Carry flag to I, indicating an error. The available characters are the ones from the starting index to the end of the string.
6. If the substring would exceed its specified maximum length, the program places only the specified maximum number of characters in lhe substring. It sets the Carry flag to 1, indicating an error.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Maximum length of substring (destination string)

Less significant byte of starting address of substring (destination string)
More significant byte of starting address of substring (destination string)

Number of bytes to copy

Starting index to copy from

Less significant byte of starting address of string (source string)
More significant byte of starting address of string (source string)

Exit Conditions

Substring contains characters copied from string. If the starting index is zero, the maximum length of the substring is zero, or the starting index is beyond the length of the string, the substring will have a length of zero and the Carry flag will be set to 1. If the substring would extend beyond the end of the string or would exceed its specified maximum length, only the available characters from the string (up to the maximum length of the substring) are copied into the substring; the Carry flag is set in this case also. If no problems occur in forming the substring, the Carry flag is cleared.

Examples

1. Data: String = 10'LET Y1 = R7 + X4' (10_{16} = 16_{10} is the length of the string)
   Maximum length of substring = 2
   Number of bytes to copy = 2
   Starting index = 5

   Result: Substring = 02'Y1' (2 is the length of the substring)
   Carry = 0, since no problems occur in forming the substring

2. Data: String = 0E'8657 POWELL ST' ($0E_{16}$ = 14_{10} is the length of the string)
   Maximum length of substring = 10_{16} = 16_{10}
   Number of bytes to copy = $0D_{16}$ = 13_{10}
   Starting index = 06

   Result: Substring = 09'POWELL ST' (09 is the length of the substring)
   Carry = 1, since there were not enough characters available in the string to provide the specified number of bytes to copy.

3. Data: String = 16'9414 HEGENBERGER DRIVE' (16_{16} = 22_{10} is the length of the string)
   Maximum length of substring = 10_{16} = 16_{10}
   Number of bytes to copy = 11_{16} = 17_{10}
   Starting index = 06

   Result: Substring = 10'HEGENBERGER DRIV' (10_{16} = 16_{10} is the length of the substring)
   Carry = 1, since the number of bytes to copy exceeded the maximum length of the substring

```
;       Title               Copy a substring from a string              ;
;       Name:               Copy                                        ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:            Copy a substring from a string given a starting ;
;                           index and the number of bytes.              ;
;                                                                       ;
;       Entry:              TOP OF STACK                                ;
;                             Low byte of return address,               ;
;                             High byte of return address,              ;
;                             Maximum length of destination string,     ;
;                             Low byte of destination string address,   ;
;                             High byte of destination string address,  ;
;                             Number of bytes to copy,                  ;
;                             Starting index to copy from,              ;
;                             Low byte of source string address,        ;
;                             High byte of source string address        ;
;                                                                       ;
;                             A string is a maximum of 255 bytes long plus ;
;                             a length byte which precedes it.          ;
;                                                                       ;
;       Exit:               Destination string := The substring from the ;
;                           string.                                     ;
;                           if no errors then                           ;
;                             CARRY := 0                                ;
;                           else                                        ;
;                             begin                                     ;
;                               the following conditions cause an       ;
;                               error and the CARRY flag = 1.           ;
;                               if (index = 0) or (maxlen = 0) or       ;
;                                   (index > length(sstrg) then         ;
;                                 the destination string will have a zero ;
;                                 length.                               ;
;                               if (index + count) > length(sstrg)) then ;
;                                 the destination string becomes everything ;
;                                 from index to the end of source string. ;
;                             END;                                      ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;                                                                       ;
;       Time:               Approximately (36 * count) cycles plus 200  ;
;                           cycles overhead.                            ;
;                                                                       ;
;       Size:               Program 173 bytes                           ;
;                           Data      6 bytes plus                      ;
;                                     4 bytes in page zero              ;
;                                                                       ;
;                                                                       ;


;EQUATES
DSTRG   .EQU    0D0H                ;PAGE ZERO POINTER TO DESTINATION STRING
SSTRG   .EQU    0D2H                ;PAGE ZERO POINTER TO SOURCE STRING
```

```
COPY:
        ;GET RETURN ADDRESS
        PLA
        TAY                         ;SAVE LOW BYTE
        PLA
        TAX                         ;SAVE HIGH BYTE

        ;GET MAXIMUM LENGTH OF DESTINATION STRING
        PLA
        STA     MAXLEN

        ;GET STARTING ADDRESS OF DESTINATION STRING
        PLA
        STA     DSTRG               ;SAVE LOW BYTE
        PLA
        STA     DSTRG+1             ;SAVE HIGH BYTE

        ;GET NUMBER OF BYTES TO COPY
        PLA
        STA     COUNT

        ;GET STARTING INDEX OF SUBSTRING
        PLA
        STA     INDEX

        ;GET STARTING ADDRESS OF SOURCE STRING
        PLA
        STA     SSTRG               ;SAVE LOW BYTE (NOTE SSTRG=SOURCE STRING)
        PLA
        STA     SSTRG+1             ;SAVE HIGH BYTE

        ;RESTORE RETURN ADDRESS
        TXA
        PHA                         ;RESTORE HIGH BYTE
        TYA
        PHA                         ;RESTORE LOW BYTE

        ;INITIALIZE LENGTH OF DESTINATION STRING AND THE ERROR FLAG TO 0
        LDA     #0
        STA     DLEN                ;LENGTH OF DESTINATION STRING IS ZERO
        STA     CPYERR              ;ASSUME NO ERRORS

        ;CHECK FOR ZERO BYTES TO COPY OR ZERO MAXIMUM SUBSTRING LENGTH
        LDA     COUNT
        BEQ     OKEXIT              ;BRANCH IF ZERO BYTES TO COPY, NO ERROR
                                    ; DSTRG WILL JUST HAVE ZERO LENGTH
        LDA     MAXLEN
        BEQ     EREXIT              ;ERROR EXIT IF SUBSTRING HAS ZERO
                                    ;  MAXIMUM LENGTH.

        LDA     INDEX
        BEQ     EREXIT              ;ERROR EXIT IF STARTING INDEX IS ZERO

        ;CHECK IF THE SOURCE STRING REACHES THE STARTING INDEX
        ;IF NOT, EXIT
        LDY     #0
```

```
        LDA     (SSTRG),Y       ;GET LENGTH OF SOURCE STRING
        STA     SLEN            ;SAVE SOURCE LENGTH
        CMP     INDEX           ;COMPARE TO STARTING INDEX
        BCC     EREXIT          ;ERROR EXIT IF INDEX IS TOO LARGE

        ;CHECK THAT WE DO NOT COPY BEYOND THE END OF THE SOURCE STRING
        ;IF INDEX + COUNT - 1 > LENGTH(SSTRG) THEN
        ; COUNT := LENGTH(SSTRG) - INDEX + 1;
        LDA     INDEX
        CLC
        ADC     COUNT
        BCS     RECALC          ;BRANCH IF INDEX + COUNT > 255
        TAX
        DEX
        CPX     SLEN
        BCC     CNT1OK          ;BRANCH IF INDEX + COUNT - 1 < LENGTH(SSTRG)
        BEQ     CNT1OK          ;BRANCH IF EQUAL

        ;THE CALLER ASKED FOR TOO MANY CHARACTERS JUST RETURN EVERYTHING
        ; BETWEEN INDEX AND THE END OF THE SOURCE STRING.
        ; SO SET COUNT := LENGTH(SSTRG) - INDEX + 1;
RECALC:
        LDA     SLEN            ;RECALCULATE COUNT
        SEC
        SBC     INDEX
        STA     COUNT
        INC     COUNT           ;COUNT := LENGTH(SSTRG) - INDEX + 1
        LDA     #0FFH
        STA     CPYERR          ;INDICATE A TRUNCATION OF THE COUNT

        ;CHECK IF THE COUNT IS LESS THAN OR EQUAL TO THE MAXIMUM LENGTH OF THE
        ; DESTINATION STRING. IF NOT, THEN SET COUNT TO THE MAXIMUM LENGTH
        ;  IF COUNT > MAXLEN THEN
        ;     COUNT := MAXLEN
CNT1OK:
        LDA     COUNT           ;IS COUNT > MAXIMUM SUBSTRING LENGTH ?
        CMP     MAXLEN
        BCC     CNT2OK          ;BRANCH IF COUNT < MAX LENGTH
        BEQ     CNT2OK          ;BRANCH IF COUNT = MAX LENGTH
        LDA     MAXLEN
        STA     COUNT           ;ELSE COUNT := MAXLEN
        LDA     #0FFH
        STA     CPYERR          ;INDICATE DESTINATION STRING OVERFLOW

        ;EVERYTHING IS SET UP SO MOVE THE SUBSTRING TO DESTINATION STRING
CNT2OK:
        LDX     COUNT           ;REGISTER X WILL BE THE COUNTER
        BEQ     EREXIT          ;ERROR EXIT IF COUNT IS ZERO
        LDA     #1              ;START WITH FIRST CHARACTER OF DESTINATION
        STA     DLEN            ;DLEN IS RUNNING INDEX FOR DESTINATION
                                ;INDEX IS RUNNING INDEX FOR SOURCE
MVLP:
        LDY     INDEX
        LDA     (SSTRG),Y       ;GET NEXT SOURCE CHARACTER
        LDY     DLEN
        STA     (DSTRG),Y       ;MOVE NEXT CHARACTER TO DESTINATION
```

```
        INC     INDEX           ;INCREMENT SOURCE INDEX
        INC     DLEN            ;INCREMENT DESTINATION INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     MVLP            ;CONTINUE UNTIL COUNTER = 0
        DEC     DLEN            ;SUBSTRING LENGTH=FINAL DESTINATION INDEX - 1
        LDA     CPYERR          ;CHECK FOR ANY ERRORS
        BNE     EREXIT          ;BRANCH IF A TRUNCATION OR STRING OVERFLOW

        ;GOOD EXIT
OKEXIT:
        CLC
        BCC     EXIT

        ;ERROR EXIT
EREXIT:
        SEC

        ;STORE LENGTH BYTE IN FRONT OF SUBSTRING
EXIT:
        LDA     DLEN
        LDY     #0
        STA     (DSTRG),Y       ;SET LENGTH OF DESTINATION STRING
        RTS

;
;DATA SECTION
SLEN:   .BLOCK  1               ;LENGTH OF SOURCE STRING
DLEN:   .BLOCK  1               ;LENGTH OF DESTINATION STRING
MAXLEN: .BLOCK  1               ;MAXIMUM LENGTH OF DESTINATION STRING
COUNT:  .BLOCK  1               ;SEARCH COUNTER
INDEX:  .BLOCK  1               ;CURRENT INDEX INTO STRING
CPYERR: .BLOCK  1               ;COPY ERROR FLAG

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC0804:
        LDA     SADR+1  ;PUSH ADDRESS OF SOURCE STRING
        PHA
        LDA     SADR
        PHA
        LDA     IDX     ;PUSH STARTING INDEX FOR COPYING
        PHA
        LDA     CNT     ;PUSH NUMBER OF CHARACTERS TO COPY
        PHA
        LDA     DADR+1  ;PUSH ADDRESS OF DESTINATION STRING
        PHA
        LDA     DADR
        PHA
        LDA     MXLEN   ;PUSH MAXIMUM LENGTH OF DESTINATION STRING
        PHA
        JSR     COPY    ;COPY
```

```
        BRK                 ;RESULT OF COPYING 3 CHARACTERS STARTING AT INDEX 4
                            ;FROM THE STRING "12.345E+10" IS 3,"345"
        JMP     SC0804      ;LOOP FOR MORE TESTING

;
;DATA SECTION
IDX     .BYTE   4                   ;STARTING INDEX FOR COPYING
CNT     .BYTE   3                   ;NUMBER OF CHARACTERS TO COPY
MXLEN   .BYTE   20H                 ;MAXIMUM LENGTH OF DESTINATION STRING
SADR    .WORD   SSTG
DADR    .WORD   DSTG
SSTG    .BYTE   0AH                 ;LENGTH OF STRING
        .BYTE   "12.345E+10                   " ;32 BYTE MAX LENGTH
DSTG    .BYTE   0                   ;LENGTH OF SUBSTRING
        .BYTE   "                             " ;32 BYTE MAX LENGTH

        .END    ;PROGRAM
```

Delete a Substring from a String (DELETE) 8E

Deletes a substring from a string, given a starting index and a length. The string is a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length. The Carry flag is cleared if the deletion can be performed as specified. The Carry flag is set if the starting index is zero or beyond the length of the string; the string is left unchanged in either case. If the deletion extends beyond the end of the string, the Carry flag is set (to 1) and only the characters from the starting index to the end of the string are deleted.

Procedure: The program exits immediately if the starting index or the number of bytes to delete is zero. It also exits if the starting index is beyond the length of the string. If none of these conditions holds, the program checks to see if the string extends beyond the area to be deleted. If it does not, the program simply truncates the string by setting the new length to the starting index minus 1. If it does, the program compacts the resulting string by moving the bytes above the deleted area down. The program then determines the new string's length and exits with the Carry cleared if the specified number of bytes were deleted and set to 1 if any errors occurred.

Registers Used: All

Execution Time: Approximately

36 * NUMBER OF BYTES MOVED DOWN + 165

where NUMBER OF BYTES MOVED DOWN is zero if the string can be truncated and is STRING LENGTH – STARTING INDEX – NUMBER OF BYTES TO DELETE + 1 if the string must be compacted.

Examples

1. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   NUMBER OF BYTES TO DELETE = 08

Since there are exactly eight bytes left in the string starting at index 19_{16}, all the routine must do is truncate the string. This takes

36 * 0 + 165 = 165 cycles.

2. STRING LENGTH = 40_{16} (64_{10})
   STARTING LENGTH = 19_{16} (25_{10})
   NUMBER OF BYTES TO DELETE = 08

Since there are 20_{16} (32_{16}) bytes above the truncated area, the routine must move them down eight positions. The execution time is

36 * 32 + 165 = 1152 + 165 = 1317 cycles.

Program Size: 139 bytes

Data Memory Required: Five bytes anywhere in RAM plus two bytes on page 0. The five bytes anywhere in RAM hold the length of the string (one byte at address SLEN), the search counter (one byte at address COUNT), an index into the string (one byte at address INDEX), the source index for use during the move (one byte at address SIDX), and an error flag (one byte at address DELERR). The two bytes on page 0 hold a pointer to the string (starting at address STRG, $00D0_{16}$ in the listing).

Special Cases:

1. If the number of bytes to delete is zero, the program exits with the Carry flag cleared (no errors) and the string unchanged.

2. If the string does not even extend to the specified starting index, the program exits with the Carry flag set to 1 (error indicated) and the string unchanged.

3. If the number of bytes to delete exceeds the number available, the program deletes all bytes from the starting index to the end of the string and exits with the Carry flag set to 1 (error indicated).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Number of bytes to delete

Starting index to delete from

Less significant byte of starting address of string

More significant byte of starting address of string

Exit Conditions

Substring deleted from string. If no errors occur, the Carry flag is cleared. If the starting index is zero or beyond the length of the string, the Carry flag is set and the string is unchanged. If the number of bytes to delete would go beyond the end of the string, the Carry flag is set and the characters from the starting index to the end of the string are deleted.

Examples

1. Data: String = 1E'SALES FOR MARCH AND APRIL 1980' ($1E_{16} = 30_{10}$ is the length of the string)

   Number of bytes to delete = $0A_{16} = 10_{10}$

   Starting index to delete from = $10_{16} = 16_{10}$

   Result: String = 14'SALES FOR MARCH 1980' ($14_{16} = 20_{10}$ is the length of the string with ten bytes deleted starting with the 16th character — the deleted material is 'AND APRIL ').

   Carry = 0, since no problems occurred in the deletion.

2. Data: String = 28'THE PRICE IS $3.00 ($2.00 BEFORE JUNE 1)' ($28_{16} = 40_{10}$ is the length of the string).

   Number of bytes to delete = $30_{16} = 48_{10}$

   Starting index to delete from = $13_{16} = 19_{10}$

   Result: String = 12'THE PRICE IS $3.00' ($12_{16} = 18_{10}$ is the length of the string with all remaining bytes deleted).

   Carry = 1, since there were not as many bytes left in the string as were supposed to be deleted.

```
;       Title           Delete a substring from a string          ;
;       Name:           Delete                                    ;
;                                                                 ;
;                                                                 ;
;                                                                 ;
;       Purpose:        Delete a substring from a string given a  ;
;                       starting index and a length.              ;
;                                                                 ;
;       Entry:          TOP OF STACK                              ;
;                         Low byte of return address,             ;
;                         High byte of return address,            ;
;                         Number of bytes to delete (count),      ;
;                         Starting index to delete from (index),  ;
;                         Low byte of string address,             ;
;                         High byte of string address             ;
```

```
;                                                                     ;
;                          A string is a maximum of 255 bytes long plus ;
;                          a length byte which precedes it.           ;
;                                                                     ;
;        Exit:             Substring deleted.                         ;
;                          if no errors then                          ;
;                            CARRY := 0                               ;
;                          else                                       ;
;                            begin                                    ;
;                              the following conditions cause an      ;
;                              error with the CARRY flag = 1.         ;
;                              if (index = 0) or (index > length(string)) ;
;                                then do not change the string        ;
;                              if count is too large then             ;
;                                delete only the characters from      ;
;                                index to the end of the string       ;
;                            end;                                     ;
;                                                                     ;
;        Registers used: All                                          ;
;                                                                     ;
;        Time:             Approximately 36 * (LENGTH(STRG)-INDEX-COUNT+1) ;
;                          plus 165 cycles overhead.                  ;
;                                                                     ;
;        Size:             Program 139 bytes                          ;
;                          Data      5 bytes plus                     ;
;                                    2 bytes in page zero             ;
;                                                                     ;
;                                                                     ;


;EQUATES
STRG    .EQU    0D0H            ;PAGE ZERO POINTER TO SOURCE STRING

DELETE:
        ;GET RETURN ADDRESS
        PLA
        TAY                     ;SAVE LOW BYTE
        PLA
        TAX                     ;SAVE HIGH BYTE

;GET NUMBER OF BYTES TO DELETE
PLA
STA     COUNT

;GET STARTING INDEX DELETION
PLA
STA     INDEX

;GET STARTING ADDRESS OF STRING
PLA
STA     STRG            ;SAVE LOW BYTE
PLA
STA     STRG+1          ;SAVE HIGH BYTE

;RESTORE RETURN ADDRESS
TXA
```

```
        PHA                     ;RESTORE HIGH BYTE
        TYA
        PHA                     ;RESTORE LOW BYTE

        ;INITIALIZE ERROR INDICATOR (DELERR) TO 0
        ;GET STRING LENGTH
        LDY     #0
        STY     DELERR
        LDA     (STRG),Y        ;GET LENGTH OF STRING
        STA     SLEN            ;SAVE STRING LENGTH

        ;CHECK FOR A NON ZERO COUNT AND INDEX
        LDA     COUNT
        BEQ     OKEXIT          ;GOOD EXIT IF NOTHING TO DELETE

        LDA     INDEX
        BEQ     EREXIT          ;ERROR EXIT IF STARTING INDEX = 0

        ;CHECK FOR STARTING INDEX WITHIN THE STRING
        ; EXIT IF IT IS NOT
        LDA     SLEN            ;IS INDEX WITHIN THE STRING ?
        CMP     INDEX
        BCC     EREXIT          ;NO, TAKE ERROR EXIT

        ;BE SURE THE NUMBER OF CHARACTERS REQUESTED TO BE DELETED ARE PRESENT
        ; IF NOT THEN ONLY DELETE FROM THE INDEX TO THE END OF THE STRING
        LDA     INDEX
        CLC
        ADC     COUNT
        BCS     TRUNC           ;TRUNCATE IF INDEX + COUNT > 255
        STA     SIDX            ;SAVE INDEX + COUNT AS THE SOURCE INDEX
        TAX                     ;X = INDEX + COUNT
        DEX
        CPX     SLEN
        BCC     CNTOK           ;BRANCH IF INDEX + COUNT - 1 < LENGTH(SSTRG)
                                ;ELSE JUST TRUNCATE THE STRING
        BEQ     TRUNC           ;TRUNCATE BUT NO ERROR (EXACTLY ENOUGH
                                ; CHARACTERS)
        LDA     #0FFH
        STA     DELERR          ;INDICATE ERROR - NOT ENOUGH CHARACTERS TO
                                ; DELETE

        ;TRUNCATE THE STRING - NO COMPACTING NECESSARY
TRUNC:
        LDX     INDEX           ;STRING LENGTH = STARTING INDEX - 1
        DEX
        STX     SLEN
        LDA     DELERR
        BEQ     OKEXIT          ;GOOD EXIT
        BNE     EREXIT          ;ERROR EXIT

        ;DELETE THE SUBSTRING BY COMPACTING
        ; MOVE ALL CHARACTERS ABOVE THE DELETED AREA DOWN
CNTOK:
        ;CALCULATE NUMBER OF CHARACTERS TO MOVE (SLEN - SIDX + 1)
```

```
        LDA     SLEN            ;GET STRING LENGTH
        SEC
        SBC     SIDX            ;SUBTRACT STARTING INDEX
        TAX
        INX                     ;ADD 1 TO INCLUDE LAST CHARACTER
        BEQ     OKEXIT          ;BRANCH IF COUNT = 0

MVLP:
        LDY     SIDX
        LDA     (STRG),Y        ;GET NEXT CHARACTER
        LDY     INDEX
        STA     (STRG),Y        ;MOVE IT DOWN
        INC     INDEX           ;INCREMENT DESTINATION INDEX
        INC     SIDX            ;INCREMENT SOURCE INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     MVLP            ;CONTINUE UNTIL COUNTER = 0
        LDX     INDEX
        DEX                     ;STRING LENGTH = FINAL DESTINATION INDEX - 1
        STX     SLEN

        ;GOOD EXIT
OKEXIT:
        CLC
        BCC     EXIT

        ;ERROR EXIT
EREXIT:
        SEC

EXIT:
        LDA     SLEN
        LDY     #0
        STA     (STRG),Y        ;SET LENGTH OF STRING
        RTS

;
;DATA
SLEN:   .BLOCK  1               ;LENGTH OF SOURCE STRING
COUNT:  .BLOCK  1               ;SEARCH COUNTER
INDEX:  .BLOCK  1               ;CURRENT INDEX INTO STRING
SIDX:   .BLOCK  1               ;SOURCE INDEX DURING MOVE
DELERR: .BLOCK  1               ;DELETE ERROR FLAG

;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;


SC0805:
        LDA     SADR+1  ;PUSH STRING ADDRESS
        PHA
        LDA     SADR
        PHA
        LDA     IDX     ;PUSH STARTING INDEX FOR DELETION
```

```
        PHA
        LDA     CNT     ;PUSH NUMBER OF CHARACTERS TO DELETE
        PHA
        JSR     DELETE  ;DELETE
        BRK             ;RESULT OF DELETING 4 CHARACTERS STARTING AT INDEX 1
                        ; FROM "JOE HANDOVER" IS "HANDOVER"
        JMP     SC0805  ;LOOP FOR ANOTHER TEST

;
;DATA SECTION
IDX     .BYTE   1               ;INDEX TO START OF DELETION
CNT     .BYTE   4               ;NUMBER OF CHARACTERS TO DELETE
SADR    .WORD   SSTG
SSTG    .BYTE   12              ;LENGTH OF STRING
        .BYTE   "JOE HANDOVER"

        .END    ;PROGRAM
```

Insert a Substring into a String (INSERT) 8F

Inserts a substring into a string, given a starting index. The string and substring are both a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length. The Carry flag is cleared if the insertion can be accomplished with no problems. The Carry flag is set if the starting index is zero or beyond the length of the string. In the second case, the substring is concatenated to the end of the string. The Carry flag is also set if the string with the insertion would exceed a specified maximum length; in that case, the program inserts only enough of the substring to give the string its maximum length.

Procedure: The program exits immediately if the starting index is zero or if the length of the substring is zero. If neither of these conditions holds, the program checks to see if the insertion would produce a string longer than the maximum. If it would, the program truncates the substring. The program then checks to see if the starting index is within the string. If it is not, the program simply concatenates the substring by moving it to the memory locations immediately after the end of the string. If the starting index is within the string, the program must first open a space for the insertion by moving the remaining characters up in memory. This move must start at the highest address to avoid writing over any data. Finally, the program can move the substring into the open area. The program then determines the new string length and exits with the Carry flag set appropriately (to 0 if no problems occurred and to 1 if the starting index was zero, the substring had to be truncated, or the starting index was beyond the length of the string).

Registers Used: All

Execution Time: Approximately 36 * NUMBER OF BYTES MOVED + 36 * NUMBER OF BYTES INSERTED + 207

NUMBER OF BYTES MOVED is the number of bytes that must be moved to open up space for the insertion. If the starting index is beyond the end of the string, this is zero since the substring is simply concatenated to the string. Otherwise, this is STRING LENGTH − STARTING INDEX + 1, since the bytes at or above the starting index must be moved.

NUMBER OF BYTES INSERTED is the length of the substring if no truncation occurs. It is the maximum length of the string minus its current length if inserting the substring would produce a string longer than the maximum.

Examples

1. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   MAXIMUM LENGTH = 30_{16} (48_{10})
   SUBSTRING LENGTH = 06

   That is, we want to insert a substring six bytes long, starting at the 25th character. Since there are eight bytes that must be moved up ($20_{16} - 19_{16} + 1$ = NUMBER OF BYTES MOVED) and six bytes that must be inserted, the execution time is approximately

   $36 * 8 + 36 * 6 + 207 = 288 + 216 + 207 = 711$ cycles.

2. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   MAXIMUM LENGTH = 24_{16} (36_{10})
   SUBSTRING LENGTH = 06

As opposed to Example 1, here only four bytes of the substring can be inserted without exceeding the maximum length of the string. Thus NUMBER OF BYTES MOVED = 8 and NUMBER OF BYTES INSERTED = 4. The execution time is approximately

$$36 * 8 + 36 * 4 + 207 = 288 + 144 + 207 = 639 \text{ cycles.}$$

Program Size: 212 bytes

Data Memory Required: Seven bytes anywhere in RAM plus four bytes on page 0. The seven bytes anywhere in RAM hold the length of the string (one byte at address SLEN), the length of the substring (one byte at address SUBLEN), the maximum length of the string (one byte at address MAXLEN), the current index into the string (one byte at address INDEX), running indexes for use during the move (one byte at address SIDX and one byte at address DIDX), and an error flag (one byte at address INSERR). The four bytes on page 0 hold pointers to the substring (two bytes starting at address SUBSTG, $00D0_{16}$ in the listing) and the string (two bytes starting at address STRG, $00D2_{16}$ in the listing).

Special Cases:

1. If the length of the substring (the insertion) is zero, the program exits with the Carry flag cleared (no error) and the string unchanged.

2. If the starting index for the insertion is zero (i.e., the insertion begins in the length byte), the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

3. If the string with the substring inserted exceeds the specified maximum length, the program inserts only enough characters to reach the maximum length. The Carry flag is set to 1 to indicate that the insertion has been truncated.

4. If the starting index of the insertion is beyond the end of the string, the program concatenates the insertion at the end of the string and indicates an error by setting the Carry flag to 1.

5. If the original length of the string exceeds its specified maximum length, the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of starting address of substring

More significant byte of starting address of substring

Maximum length of string

Starting index at which to insert the substring

Less significant byte of starting address of string

More significant byte of starting address of string

Exit Conditions

Substring inserted into string. If no errors occur, the Carry flag is cleared. If the starting index is zero or the length of the substring is zero, the Carry flag is set and the string is not changed. If the starting index is beyond the length of the string, the Carry flag is set and the substring is concatenated to the end of the string. If the string with the substring inserted would exceed the specified maximum length, the Carry flag is set and only those characters from the substring which bring the string to maximum length are inserted.

Examples

1. Data: String = 0A'JOHN SMITH' ($0A_{16} = 10_{10}$ is the length of the string)
   Substring = 08'WILLIAM ' (08 is the length of the substring)
   Maximum length of string = $14_{16} = 20_{10}$
   Starting index = 06

   Result: String = 12'JOHN WILLIAM SMITH' ($12_{16} = 18_{16}$ is the length of the string with the substring inserted).
   Carry = 0, since no problems occurred in the insertion.

2. Data: String = 0A'JOHN SMITH' ($0A_{16} = 10_{10}$ is the length of the string)
   Substring = 0C'ROCKEFELLER ' ($0C_{16} = 12_{10}$ is the length of the substring)
   Maximum length of string = $14_{16} = 20_{10}$
   Starting index = 06

   Result: String = 14'JOHN ROCKEFELLESMITH' ($14_{16} = 20_{10}$ is the length of the string with as much of the substring inserted as the maximum length would allow)
   Carry = 1, since some of the substring could not be inserted without exceeding the maximum length of the string.

```
;        Title            Insert a substring into a string           ;
;        Name:            Insert                                     ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;        Purpose:         Insert a substring into a string given a   ;
;                         starting index.                            ;
;                                                                    ;
;        Entry:           TOP OF STACK                               ;
;                           Low byte of return address,              ;
;                           High byte of return address,             ;
;                           Low byte of substring address,           ;
;                           High byte of substring address,          ;
;                           Maximum length of (source) string,       ;
;                           Starting index to insert the substring,  ;
;                           Low byte of (source) string address,     ;
;                           High byte of (source) string address     ;
;                                                                    ;
;                           A string is a maximum of 255 bytes long plus ;
;                           a length byte which precedes it.         ;
;                                                                    ;
;        Exit:            Substring inserted into string.            ;
;                         if no errors then                          ;
;                           CARRY = 0                                ;
;                         else                                       ;
;                           begin                                    ;
;                             the following conditions cause the     ;
;                             CARRY flag to be set.                  ;
;                             if index = 0 then                      ;
;                               do not insert the substring          ;
;                             if length(strg) > maximum length then  ;
;                               do not insert the substring          ;
```

```
;                            if index > length(strg) then              ;
;                              concatenate substg onto the end of the  ;
;                              source string                           ;
;                            if length(strg)+length(substring) > maxlen ;
;                              then insert only enough of the substring ;
;                              to reach maximum length                 ;
;                          end;                                        ;
;                                                                      ;
;       Registers used: All                                            ;
;                                                                      ;
;       Time:           Approximately                                  ;
;                        36 * (LENGTH(STRG) - INDEX + 1) +             ;
;                        36 * (LENGTH(SUBSTG)) +                       ;
;                        207 cycles overhead.                          ;
;                                                                      ;
;       Size:           Program 214 bytes                              ;
;                       Data      7 bytes plus                         ;
;                                 4 bytes in page zero                 ;
;                                                                      ;
;                                                                      ;

;EQUATES
SUBSTG  .EQU    0D0H            ;PAGE ZERO POINTER TO SUBSTRING
STRG    .EQU    0D2H            ;PAGE ZERO POINTER TO STRING

INSERT:
        ;GET RETURN ADDRESS
        PLA
        TAY                     ;SAVE LOW BYTE
        PLA
        TAX                     ;SAVE HIGH BYTE

        ;GET STARTING ADDRESS OF SUBSTRING
        PLA
        STA     SUBSTG          ;SAVE LOW BYTE
        PLA
        STA     SUBSTG+1        ;SAVE HIGH BYTE

        ;GET MAXIMUM LENGTH OF STRING
        PLA
        STA     MAXLEN

        ;GET STARTING INDEX for insertion
        PLA
        STA     INDEX

        ;GET STARTING ADDRESS OF SOURCE STRING
        PLA
        STA     STRG            ;SAVE LOW BYTE
        PLA
        STA     STRG+1          ;SAVE HIGH BYTE

        ;RESTORE RETURN ADDRESS
        TXA
        PHA                     ;RESTORE HIGH BYTE
        TYA
```

```
        PHA                             ;RESTORE LOW BYTE

        ;ASSUME NO ERRORS
        LDA     #0
        STA     INSERR                  ;ASSUME NO ERRORS WILL BE FOUND

        ;GET SUBSTRING AND STRING LENGTHS
        ; IF LENGTH(SUBSTG) = 0 THEN EXIT BUT NO ERROR
        LDY     #0
        LDA     (STRG),Y
        STA     SLEN                    ;GET LENGTH OF STRING
        LDA     (SUBSTG),Y
        STA     SUBLEN                  ;GET LENGTH OF SUBSTRING
        BNE     IDX0
        JMP     OKEXIT                  ;EXIT IF NOTHING TO INSERT (NO ERROR)
        ;IF STARTING INDEX IS ZERO THEN ERROR EXIT
IDX0:
        LDA     INDEX
        BNE     CHKLEN                  ;BRANCH IF INDEX NOT EQUAL 0
        JMP     EREXIT                  ;ELSE ERROR EXIT

        ;CHECK THAT THE RESULTING STRING AFTER THE INSERTION FITS IN THE
        ; SOURCE STRING. IF NOT THEN TRUNCATE THE SUBSTRING AND SET THE
        ; TRUNCATION FLAG.
CHKLEN:
        LDA     SUBLEN                  ;GET SUBSTRING LENGTH
        CLC
        ADC     SLEN
        BCS     TRUNC                   ;TRUNCATE SUBSTRING IF NEW LENGTH > 255
        CMP     MAXLEN
        BCC     IDXLEN                  ;BRANCH IF NEW LENGTH < MAX LENGTH
        BEQ     IDXLEN                  ;BRANCH IF NEW LENGTH = MAX LENGTH

        ;SUBSTRING DOES NOT FIT, SO TRUNCATE IT
TRUNC:
        LDA     MAXLEN                  ;SUBSTRING LENGTH = MAXIMUM LENGTH - STRING
                                        ; LENGTH
        SEC
        SBC     SLEN
        BCC     EREXIT                  ;ERROR EXIT IF MAXIMUM LENGTH < STRING LENGTH
        BEQ     EREXIT                  ;ERROR EXIT IF SUBSTRING LENGTH IS ZERO
                                        ; (THE ORIGINAL STRING WAS TOO LONG !!)
        STA     SUBLEN
        LDA     #0FFH
        STA     INSERR                  ;INDICATE SUBSTRING WAS TRUNCATED

        ;CHECK THAT INDEX IS WITHIN THE STRING. IF NOT CONCATENATE THE
        ; SUBSTRING ONTO THE END OF THE STRING.
IDXLEN:
        LDA     SLEN                    ;GET STRING LENGTH
        CMP     INDEX                   ;COMPARE TO INDEX
        BCS     LENOK                   ;BRANCH IF STARTING INDEX IS WITHIN STRING
        LDX     SLEN                    ;ELSE JUST CONCATENATE (PLACE SUBSTRING AT
                                        ; END OF STRING)
        INX
```

```
        STX     INDEX           ;START RIGHT AFTER END OF STRING
        LDA     #0FFH
        STA     INSERR          ;INDICATE ERROR IN INSERT
        LDA     SLEN            ;ADD SUBSTRING LENGTH TO STRING LENGTH
        CLC
        ADC     SUBLEN
        STA     SLEN
        JMP     MVESUB          ;JUST PERFORM MOVE, NOTHING TO OPEN UP

        ;OPEN UP A SPACE IN SOURCE STRING FOR THE SUBSTRING BY MOVING THE
        ; CHARACTERS FROM THE END OF THE SOURCE STRING DOWN TO INDEX, UP BY
        ; THE SIZE OF THE SUBSTRING.
LENOK:
        ;CALCULATE NUMBER OF CHARACTERS TO MOVE
        ; COUNT := STRING LENGTH - STARTING INDEX + 1
        LDA     SLEN
        SEC
        SBC     INDEX
        TAX
        INX                     ;X = NUMBER OF CHARACTERS TO MOVE

        ;SET THE SOURCE INDEX AND CALCULATE THE DESTINATION INDEX
        LDA     SLEN
        STA     SIDX            ;SOURCE ENDS AT END OF ORIGINAL STRING
        CLC
        ADC     SUBLEN
        STA     DIDX            ;DESTINATION ENDS FURTHER BY SUBSTRING LENGTH
        STA     SLEN            ;SET THE NEW LENGTH TO THIS VALUE ALSO

OPNLP:
        LDY     SIDX
        LDA     (STRG),Y        ;GET NEXT CHARACTER
        LDY     DIDX
        STA     (STRG),Y        ;MOVE IT UP IN MEMORY
        DEC     SIDX            ;DECREMENT SOURCE INDEX
        DEC     DIDX            ;DECREMENT DESTINATION INDEX
        DEX                     ;DECREMENT COUNTER
        BNE     OPNLP           ;CONTINUE UNTIL COUNTER = 0

        ;MOVE THE SUBSTRING INTO THE OPEN AREA
MVESUB:
        LDA     #1
        STA     SIDX            ;START AT ONE IN THE SUBSTRING
                                ;START AT INDEX IN THE STRING
        LDX     SUBLEN          ;X = NUMBER OF CHARACTERS TO MOVE

MVELP:
        LDY     SIDX
        LDA     (SUBSTG),Y      ;GET NEXT CHARACTER
        LDY     INDEX
        STA     (STRG),Y        ;STORE CHARACTER
        INC     SIDX            ;INCREMENT SUBSTRING INDEX
        INC     INDEX           ;INCREMENT STRING INDEX
        DEX                     ;DECREMENT COUNT
        BNE     MVELP           ;CONTINUE UNTIL COUNTER = 0
        LDA     INSERR          ;GET ERROR FLAG
```

```
        BNE     EREXIT              ;BRANCH IF SUBSTRING WAS TRUNCATED

OKEXIT:
        CLC                         ;NO ERROR
        BCC     EXIT

EREXIT:
        SEC                         ;ERROR EXIT

EXIT:
        LDA     SLEN
        LDY     #0
        STA     (STRG),Y            ;SET NEW LENGTH OF STRING
        RTS
;
;DATA SECTION
SLEN:   .BLOCK  1                   ;LENGTH OF STRING
SUBLEN: .BLOCK  1                   ;LENGTH OF SUBSTRING
MAXLEN: .BLOCK  1                   ;MAXIMUM LENGTH OF STRING
INDEX:  .BLOCK  1                   ;CURRENT INDEX INTO STRING
SIDX:   .BLOCK  1                   ;A RUNNING INDEX
DIDX:   .BLOCK  1                   ;A RUNNING INDEX
INSERR: .BLOCK  1                   ;FLAG USED TO INDICATE IF AN ERROR

;                                                                  ;
;                                                                  ;
;       SAMPLE EXECUTION:                                          ;
;                                                                  ;
;                                                                  ;


SC0806:
        LDA     SADR+1   ;PUSH ADDRESS OF SOURCE STRING
        PHA
        LDA     SADR
        PHA
        LDA     IDX      ;PUSH STARTING INDEX FOR INSERTION
        PHA
        LDA     MXLEN    ;PUSH MAXIMUM LENGTH OF SOURCE STRING
        PHA
        LDA     SUBADR+1 ;PUSH ADDRESS OF THE SUBSTRING
        PHA
        LDA     SUBADR
        PHA
        JSR     INSERT   ;INSERT
        BRK              ;RESULT OF INSERTING "-" INTO "123456" AT
                         ; INDEX 1 IS "-123456"
        JMP     SC0806   ;LOOP FOR ANOTHER TEST

;
;DATA SECTION
IDX     ..BYTE  1                   ;INDEX TO START INSERTION
MXLEN   .BYTE   20H                 ;MAXIMUM LENGTH OF DESTINATION
SADR    .WORD   STG                 ;STARTING ADDRESS OF STRING
SUBADR  .WORD   SSTG                ;STARTING ADDRESS OF SUBSTRING
STG     .BYTE   06H                 ;LENGTH OF STRING
```

```
        .BYTE    "123456                          " ;32 BYTE MAX LENGTH
SSTG    .BYTE    1               ;LENGTH OF SUBSTRING
        .BYTE    "-                               " ;32 BYTE MAX LENGTH

        .END     ;PROGRAM
```

Adds the elements of a byte-length array, producing a 16-bit sum. The size of the array is specified and is a maximum of 255 bytes.

Procedure: The program clears both bytes of the sum initially. It then adds the elements successively to the less significant byte of the sum, starting with the element at the highest address. Whenever an addition produces a carry, the program increments the more significant byte of the sum.

Registers Used: All

Execution Time: Approximately 16 cycles per byte plus 39 cycles overhead. If, for example, (X) = $1A_{16}$ = 26_{10}, the execution time is approximately

$16 * 26 + 39 = 416 + 39 = 455$ cycles.

Program Size: 30 bytes

Data Memory Required: Two bytes on page 0 to hold a pointer to the array (starting at address ARYADR, $00D0_{16}$ in the listing).

Special Case: An array size of zero causes an immediate exit with the sum equal to zero.

Entry Conditions

(A) = More significant byte of starting address of array

(Y) = Less significant byte of starting address of array

(X) = Size of array in bytes

Exit Conditions

(A) = More significant byte of sum

(Y) = Less significant byte of sum

Example

Data: Size of array (in bytes) = (X) = 08

Array elements

$F7_{16} = 247_{10}$
$23_{16} = 35_{10}$
$31_{16} = 49_{10}$
$70_{16} = 112_{10}$
$5A_{16} = 90_{10}$
$16_{16} = 22_{10}$
$CB_{16} = 203_{10}$
$E1_{16} = 225_{10}$

Result: Sum = $03D7_{16} = 983_{10}$

(A) = more significant byte of sum = 03_{16}

(Y) = less significant byte of sum = $D7_{16}$

```
;       Title               8 BIT ARRAY SUMMATION                          ;
;       Name:               ASUM8                                          ;
;                                                                          ;
;                                                                          ;

;                                                                          ;
;       Purpose:            SUM the data of an array, yielding a 16 bit    ;
;                           result. Maximum size is 255.                   ;
;                                                                          ;
;       Entry:              Register A = High byte of starting array address;
;                           Register Y = Low byte of starting array address ;
;                           Register X = Size of array in bytes            ;
;                                                                          ;
;       Exit:               Register A = High byte of sum                  ;
;                           Register Y = Low byte of sum                   ;
;                                                                          ;
;       Registers used: All                                                ;
;                                                                          ;
;       Time:               Approximately 16 cycles per byte plus          ;
;                           39 cycles overhead.                            ;
;                                                                          ;
;       Size:               Program 30 bytes                               ;
;                           Data     2 bytes in page zero                  ;
;                                                                          ;
;                                                                          ;

;
;EQUATES SECTION
ARYADR: .EQU    0D0H            ;PAGE ZERO POINTER TO ARRAY

ASUM8:
        ;
        ;STORE STARTING ADDRESS
        STY     ARYADR
        STA     ARYADR+1

        ;DECREMENT STARTING ADDRESS BY 1 FOR EFFICIENT PROCESSING
        TYA                     ;GET LOW BYTE OF STARTING ADDRESS
        BNE     ASUM81          ;IS LOW BYTE ZERO ?
        DEC     ARYADR+1        ;YES, BORROW FROM HIGH BYTE
ASUM81: DEC     ARYADR          ;ALWAYS DECREMENT LOW BYTE

        ;EXIT IF LENGTH OF ARRAY IS ZERO
        TXA
        TAY
        BEQ     EXIT            ;EXIT IF LENGTH IS ZERO

        ;INITIALIZATION
        LDA     #0              ;INITIALIZE SUM TO 0
        TAX

        ;SUMMATION LOOP
SUMLP:
        CLC
        ADC     (ARYADR),Y      ;ADD NEXT BYTE TO LSB OF SUM
        BCC     DECCNT
        INX                     ;INCREMENT MSB OF SUM IF A CARRY OCCURS
```

```
DECCNT:
        DEY                     ;DECREMENT COUNT
        BNE     SUMLP           ;CONTINUE UNTIL REGISTER Y EQUALS 0

EXIT:
        TAY                     ;REGISTER Y = LOW BYTE OF SUM
        TXA                     ;REGISTER A = HIGH BYTE OF SUM
        RTS


;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION                                        ;
;                                                               ;
;                                                               ;

;
SC0901:
        LDY     BUFADR          ;Y IS LOW BYTE OF BUFFER ADDRESS
        LDA     BUFADR+1        ;A IS HIGH BYTE OF BUFFER ADDRESS
        LDX     BUFSZ           ;X IS SIZE OF BUFFER
        JSR     ASUM8
        BRK                     ;SUM OF THE INITIAL TEST DATA IS 07F8 HEX,
                                ; REGISTER A = 07, REGISTER Y = F8H
        JMP     SC0901

;TEST DATA, CHANGE FOR OTHER VALUES
SIZE    .EQU    010H            ;SIZE OF BUFFER
BUFADR: .WORD   BUF             ;STARTING ADDRESS OF BUFFER

BUFSZ:  .BYTE   SIZE            ;SIZE OF BUFFER

BUF:    .BYTE   00H             ;BUFFER
        .BYTE   11H             ;DECIMAL ELEMENTS ARE 0,17,34,51,68
        .BYTE   22H             ; 85,102,119,136,153,170,187,204
        .BYTE   33H             ; 221,238,255
        .BYTE   44H
        .BYTE   55H
        .BYTE   66H
        .BYTE   77H
        .BYTE   88H
        .BYTE   99H
        .BYTE   0AAH
        .BYTE   0BBH
        .BYTE   0CCH
        .BYTE   0DDH
        .BYTE   0EEH
        .BYTE   0FFH            ;SUM = 07F8 (2040 DECIMAL)

        .END                    ;PROGRAM
```

16-Bit Array Summation (ASUM16) 9B

Adds the elements of a word-length array, producing a 24-bit sum. The size of the array is specified and is a maximum of 255 16-bit words. The 16-bit elements are stored in the usual 6502 style with the less significant byte first.

Procedure: The program clears a 24-bit accumulator in three bytes of memory and then adds the elements to the memory accumulator, starting at the lowest address. The most significant byte of the memory accumulator is incremented each time the addition of the more significant byte of an element and the middle byte of the sum produces a carry. If the array occupies more than one page of memory, the program must increment the more significant byte of the array pointer before proceeding to the second page.

Registers Used: All

Execution Time: Approximately 43 cycles per byte plus 46 cycles overhead. If, for example, (X) $= 12_{16} = 18_{10}$, the execution time is approximately

$43 * 18 + 46 = 774 + 46 = 820$ cycles.

Program Size: 60 bytes

Data Memory Required: Three bytes anywhere in RAM plus two bytes on page 0. The three bytes anywhere in RAM hold the memory accumulator (starting at address SUM); the two bytes on page 0 hold a pointer to the array (starting at address ARYADR, $00D0_{16}$ in the listing).

Special Case: An array size of 0 causes an immediate exit with the sum equal to zero.

Entry Conditions

(A) = More significant byte of starting address of array

(Y) = Less significant byte of starting address of array

(X) = Size of array in 16-bit words

Exit Conditions

(X) = Most significant byte of sum

(A) = Middle byte of sum

(Y) = Least significant byte of sum

Example

Data: Size of array (in 16-bit words) = (X) = 08

Array elements

$F7A1_{16} = 63{,}393_{10}$
$239B_{16} = 9{,}115_{10}$
$31D5_{16} = 12{,}757_{10}$
$70F2_{16} = 28{,}914_{10}$
$5A36_{16} = 23{,}094_{10}$
$166C_{16} = 5{,}740_{10}$
$CBF5_{16} = 52{,}213_{10}$
$E107_{16} = 57{,}607_{10}$

Result: Sum $= 03DBA1_{16} = 252{,}833_{10}$

(X) = most significant byte of sum $= 03_{16}$
(A) = middle byte of sum $= DB_{16}$
(Y) = least significant byte of sum $= A1_{16}$

```
;       Title           16 BIT ARRAY SUMMATION                      ;
;       Name:           ASUM16                                      ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Purpose:        Sum the data of an array, yielding a 24 bit ;
;                       result. Maximum size is 255 16 bit elements.;
;                                                                   ;
;       Entry:          Register A = High byte of starting array address;
;                       Register Y = Low byte of starting array address ;
;                       Register X = size of array in 16 bit elements   ;
;                                                                   ;
;       Exit:           Register X = High byte of sum               ;
;                       Register A = Middle byte of sum             ;
;                       Register Y = Low byte of sum                ;
;                                                                   ;
;       Registers used: All                                         ;
;                                                                   ;
;       Time:           Approximately 43 cycles per byte plus       ;
;                       46 cycles overhead.                         ;
;                                                                   ;
;       Size:           Program 60 bytes                            ;
;                       Data     3 bytes plus                       ;
;                                2 bytes in page zero               ;
;                                                                   ;
;                                                                   ;

;
;EQUATES SECTION
ARYADR: .EQU    0D0H            ;PAGE ZERO POINTER TO ARRAY

ASUM16:
        ;
        ;STORE STARTING ADDRESS
        STY     ARYADR
        STA     ARYADR+1

        ;ZERO SUM AND INITIALIZE INDEX
        LDA     #0
        STA     SUM             ;SUM = 0
        STA     SUM+1
        STA     SUM+2
        TAY                     ;INDEX = 0

        ;EXIT IF THE ARRAY LENGTH IS ZERO
        TXA
        BEQ     EXIT

        ;SUMMATION LOOP
SUMLP:
        LDA     SUM
        CLC
        ADC     (ARYADR),Y      ;ADD LOW BYTE OF ELEMENT TO SUM
        STA     SUM
```

```
        LDA     SUM+1
        INY                     ;INCREMENT INDEX TO HIGH BYTE OF ELEMENT
        ADC     (ARYADR),Y      ;ADD HIGH BYTE WITH CARRY TO SUM
        STA     SUM+1           ;STORE IN MIDDLE BYTE OF SUM
        BCC     NXTELM
        INC     SUM+2           ;INCREMENT HIGH BYTE OF SUM IF A CARRY
NXTELM:
        INY                     ;INCREMENT INDEX TO NEXT ARRAY ELEMENT
        BNE     DECCNT
        INC     ARYADR+1        ;MOVE POINTER TO SECOND PAGE OF ARRAY
DECCNT:
        DEX                     ;DECREMENT COUNT
        BNE     SUMLP           ;CONTINUE UNTIL REGISTER X EQUALS 0

EXIT:
        LDY     SUM             ;Y=LOW BYTE
        LDA     SUM+1           ;A=MIDDLE BYTE
        LDX     SUM+2           ;X=HIGH BYTE
        RTS

;DATA SECTION
SUM:    .BLOCK  3               ;TEMPORARY 24 BIT ACCUMULATOR IN MEMORY

;
;                                                               ;
;       SAMPLE EXECUTION                                        ;
;                                                               ;
;                                                               ;
                                                                ;
;
SC0902:
        LDY     BUFADR          ;A,Y = STARTING ADDRES OF BUFFER
        LDA     BUFADR+1
        LDX     BUFSZ           ;X = BUFFER SIZE IN WORDS
        JSR     ASUM16          ;RESULT OF THE INITIAL TEST DATA IS 12570
        BRK                     ; REGISTER X = 0, REGISTER A = 31H,
                                ; REGISTER Y = 1AH
        JMP     SC0902          ;LOOP FOR MORE TESTING

;
SIZE    .EQU    010H            ;SIZE OF BUFFER IN WORDS
BUFADR: .WORD   BUF             ;STARTING ADDRESS OF BUFFER
BUFSZ:  .BYTE   SIZE            ;SIZE OF BUFFER IN WORDS
BUF:    .WORD   0               ;BUFFER
        .WORD   111
        .WORD   222
        .WORD   333
        .WORD   444
        .WORD   555
        .WORD   666
        .WORD   777
        .WORD   888
        .WORD   999
        .WORD   1010
        .WORD   1111
        .WORD   1212
```

```
        .WORD    1313
        .WORD    1414
        .WORD    1515            ;SUM = 12570 = 311AH

        .END     ;PROGRAM
```

Find Maximum Byte-Length Element (MAXELM)

Finds the maximum element in an array of unsigned byte-length elements. The size of the array is specified and is a maximum of 255 bytes.

Procedure: The program exits immediately (setting Carry to 1) if the array size is zero. If the size is non-zero, the program assumes that the last byte of the array is the largest and then proceeds backward through the array, comparing the supposedly largest element to the current element and retaining the larger value and its index. Finally, the program clears the Carry to indicate a valid result.

Registers Used: All

Execution Time: Approximately 15 to 23 cycles per byte plus 52 cycles overhead. The extra eight cycles are used whenever the supposed maximum and its index must be replaced by the current element and its index. If, on the average, that replacement occurs half the time, the execution time is approximately

38 * ARRAY SIZE/2 + 52 cycles.

If, for example, ARRAY SIZE = $18_{16} = 24_{10}$, the approximate execution time is

38 * 12 + 52 = 456 + 52 = 508 cycles.

Program Size: 45 bytes

Data Memory Required: One byte anywhere in RAM plus two bytes on page 0. The one byte anywhere in RAM holds the index of the largest element (at address INDEX). The two bytes on page 0 hold a pointer to the array (starting at address ARYADR, $00D0_{16}$ in the listing).

Special Cases:

1. An array size of 0 causes an immediate exit with the Carry flag set to 1 to indicate an invalid result.

2. If more than one element has the largest unsigned value, the program returns with the smallest possible index. That is, the index designates the occurrence of the maximum value closest to the starting address.

Entry Conditions

(A) = More significant byte of starting address of array

(Y) = Less significant byte of starting address of array

(X) = Size of array in bytes

Exit Conditions

(A) = Largest unsigned element

(Y) = Index to largest unsigned element

Carry = 0 if result is valid, 1 if size of array is 0 and result is meaningless.

Example

Data: Size of array (in bytes) = (X) = 08

Array elements

$35_{16} = 53_{10}$	$44_{16} = 68_{10}$
$A6_{16} = 166_{10}$	$59_{16} = 89_{10}$
$D2_{16} = 210_{10}$	$7A_{16} = 122_{10}$
$1B_{16} = 27_{10}$	$CF_{16} = 207_{10}$

Result: The largest unsigned element is element #2 ($D2_{16} = 210_{10}$)

(A) = largest element ($D2_{16}$)

(Y) = index to largest element (02)

Carry flag = 0, indicating that array size is non-zero and the result is valid

```
;       Title                 Find the maximum element in an array of unsigned;
;                             bytes.                                          ;
;       Name:                 MAXELM                                          ;
;                                                                             ;
;                                                                             ;
;                                                                             ;
;                                                                             ;
;       Purpose:              Given the starting address of an array and     ;
;                             the size of the array, find the largest element ;
;                                                                             ;
;       Entry:                Register A = High byte of starting  address     ;
;                             Register Y = Low byte of starting address       ;
;                             Register X = Size of array in bytes             ;
;                                                                             ;
;       Exit:                 If size of the array is not zero then           ;
;                               CARRY FLAG = 0                                ;
;                               Register A = Largest element                  ;
;                               Register Y = Index to that element            ;
;                                if there are duplicate values of the largest ;
;                                element, register Y will have the index      ;
;                                nearest to the first array element           ;
;                             else                                            ;
;                               CARRY flag = 1                                ;
;                                                                             ;
;       Registers used: All                                                   ;
;                                                                             ;
;       Time:                 Approximately 15 to 23 cycles per byte          ;
;                             plus 52 cycles overhead.                        ;
;                                                                             ;
;       Size:                 Program 45 bytes                                ;
;                             Data     1 byte plus                            ;
;                                      2 bytes in page zero                   ;
;                                                                             ;
;                                                                             ;

;EQUATES
ARYADR: .EQU    0D0H                ;PAGE ZERO FOR ARRAY POINTER

MAXELM:
        ;STORE STARTING ARRAY ADDRESS
        STA     ARYADR+1
        STY     ARYADR

        ;SUBTRACT 1 FROM STARTING ADDRESS TO INDEX FROM 1 TO SIZE
        TYA
        BNE     MAX1
        DEC     ARYADR+1            ;BORROW FROM HIGH BYTE IF LOW BYTE = 0
MAX1:   DEC     ARYADR              ;ALWAYS DECREMENT THE LOW BYTE

        ;TEST FOR SIZE EQUAL TO ZERO AND INITIALIZE TEMPORARIES

        TXA
        BEQ     EREXIT              ;ERROR EXIT IF SIZE IS ZERO
        TAY                         ;REGISTER Y = SIZE AND INDEX
        LDA     (ARYADR),Y          ;GET LAST BYTE OF ARRAY
        STY     INDEX               ;SAVE ITS INDEX
```

```
        DEY
        BEQ     OKEXIT          ;EXIT IF ONLY ONE ELEMENT

        ;WORK FROM THE END OF THE ARRAY TOWARDS THE BEGINNING COMPARING
        ; AGAINST THE CURRENT MAXIMUM WHICH IS IN REGISTER A
MAXLP:
        CMP     (ARYADR),Y
        BEQ     NEWIDX          ;REPLACE INDEX ONLY IF ELEMENT = MAXIMUM
        BCS     NXTBYT          ;BRANCH IF CURRENT MAXIMUM > ARY[Y]
                                ;ELSE ARY[Y] >= CURRENT MAXIMUM SO
        LDA     (ARYADR),Y      ; NEW CURRENT MAXIMUM AND
NEWIDX: STY     INDEX           ; NEW INDEX
NXTBYT:
        DEY                     ;DECREMENT TO NEXT ELEMENT
        BNE     MAXLP           ;CONTINUE

        ;
        ;EXIT
OKEXIT:
        LDY     INDEX           ;GET INDEX OF THE MAXIMUM ELEMENT
        DEY                     ;NORMALIZE INDEX TO (0,SIZE-1)
        CLC                     ;NO ERRORS
        RTS

EREXIT:
        SEC                     ;ERROR, NO ELEMENTS IN THE ARRAY
        RTS

;DATA SECTION
INDEX:  .BLOCK  1               ;INDEX OF LARGEST ELEMENT

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC0903:
        LDA     AADR+1          ;A,Y = STARTING ADDRESS OF ARRAY
        LDY     AADR
        LDX     #SZARY          ;X = SIZE OF ARRAY
        JSR     MAXELM
        BRK                     ;RESULT FOR THE INITIAL TEST DATA IS
                                ; A = FF HEX (MAXIMUM), Y=08 (INDEX TO MAXIMUM)
        JMP     SC0903          ;LOOP FOR MORE TESTING


SZARY:  .EQU    10H             ;SIZE OF ARRAY
AADR:   .WORD   ARY             ;STARTING ADDRESS OF ARRAY
ARY:    .BYTE   8
        .BYTE   7
        .BYTE   6
        .BYTE   5
```

```
.BYTE    4
.BYTE    3
.BYTE    2
.BYTE    1
.BYTE    0FFH
.BYTE    0FEH
.BYTE    0FDH
.BYTE    0FCH
.BYTE    0FBH
.BYTE    0FAH
.BYTE    0F9H
.BYTE    0F8H

.END     ;PROGRAM
```

Finds the minimum element in an array of unsigned byte-length elements. The size of the array is specified and is a maximum of 255 bytes.

Procedure: The program exits immediately, setting Carry to 1, if the array size is zero. If the size is non-zero, the program assumes that the last byte of the array is the smallest and then proceeds backward through the array, comparing the supposedly smallest element to the current element and retaining the smaller value and its index. Finally, the program clears the Carry flag to indicate a valid result.

Registers Used: All

Execution Time: Approximately 15 to 23 cycles per byte plus 52 cycles overhead. The extra eight cycles are used whenever the supposed minimum and its index must be replaced by the current element and its index. If, on the average, that replacement occurs half the time, the execution time is approximately

38 * ARRAY SIZE/2 + 52 cycles.

If, for example, ARRAY SIZE = $14_{16} = 20_{10}$, the approximate execution time is

38 * 10 + 52 = 380 + 52 = 432 cycles.

Program Size: 45 bytes

Data Memory Required: One byte anywhere in RAM plus two bytes on page 0. The one byte anywhere in RAM holds the index of the smallest element (at address INDEX). The two bytes on page 0 hold a pointer to the array (starting at address ARYADR, $00D0_{16}$ in the listing).

Special Cases:

1. An array size of 0 causes an immediate exit with the Carry flag set to 1 to indicate an invalid result.

2. If more than one element has the smallest unsigned value, the program returns with the smallest possible index. That is, the index designates the occurrence of the minimum value closest to the starting address.

Entry Conditions

(A) = More significant byte of starting address of array

(Y) = Less significant byte of starting address of array

(X) = Size of array in bytes

Exit Conditions

(A) = Smallest unsigned element

(Y) = Index to smallest unsigned element

Carry = 0 if result is valid, 1 if size of array is zero and result is meaningless.

Example

Data: Size of array (in bytes) = (X) = 08

Array elements

$35_{16} = 53_{10}$	$44_{16} = 68_{10}$
$A6_{16} = 166_{10}$	$59_{16} = 89_{10}$
$D2_{16} = 210_{10}$	$7A_{16} = 122_{10}$
$1B_{16} = 27_{10}$	$CF_{16} = 207_{10}$

Result: The smallest unsigned element is element #3 ($1B_{16} = 27_{10}$)

(A) = smallest element ($1B_{16}$)

(Y) = index to smallest element (03)

Carry flag = 0, indicating that array size is non-zero and the result is valid.

```
;       Title           Find the minimum element in an array of unsigned;
;                       bytes.                                          ;
;       Name:           MINELM                                          ;
;                                                                       ;
;                                                                       ;
                                                                        ;
;       Purpose:        Given the STARTING ADDRESS and the size of an   ;
;                       array, find the smallest element.               ;
;                                                                       ;
;       Entry:          Register A = High byte of starting  address     ;
;                       Register Y = Low byte of starting address       ;
;                       Register X = Size of array in bytes             ;
;                                                                       ;
;       Exit:           If size of the array is not zero then           ;
;                         CARRY FLAG = 0                                ;
;                         Register A = Smallest element                 ;
;                         Register Y = Index to that element            ;
;                          if there are duplicate values of the smallest;
;                          element Register Y will have the index       ;
;                          nearest to the first array element           ;
;                       else                                            ;
;                         CARRY flag = 1                                ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           Approximately 15 to 23 cycles per byte          ;
;                       plus 52 cycles overhead.                        ;
;                                                                       ;
;       Size:           Program 45 bytes                                ;
;                       Data     1 bytes plus                           ;
;                                2 bytes in page zero                   ;
;                                                                       ;
;                                                                       ;

;EQUATES
ARYADR: .EQU    0D0H            ;PAGE ZERO POINTER TO ARRAY

MINELM:
        ;STORE STARTING ARRAY ADDRESS
        STA     ARYADR+1
        STY     ARYADR

        ;DECREMENT ARRAY ADDRESS BY 1 TO INDEX FROM 1 TO SIZE
        TYA
        BNE     MIN1
        DEC     ARYADR+1        ;BORROW FROM HIGH BYTE IF LOW BYTE = 0
MIN1:   DEC     ARYADR          ;ALWAYS DECREMENT THE LOW BYTE

        ;TEST FOR SIZE EQUAL TO ZERO AND INITIALIZE TEMPORARIES

        TXA
        BEQ     EREXIT          ;ERROR EXIT IF SIZE IS ZERO
        TAY                     ;REGISTER Y = SIZE AND INDEX
        LDA     (ARYADR),Y      ;GET LAST BYTE OF ARRAY
        STY     INDEX           ;SAVE ITS INDEX
```

```
        DEY
        BEQ     OKEXIT              ;EXIT IF ONLY ONE ELEMENT

        ;WORK FROM THE END OF THE ARRAY TOWARDS THE BEGINNING COMPARING
        ; AGAINST THE CURRENT MINIMUM WHICH IS IN REGISTER A
MINLP:
        CMP     (ARYADR),Y
        BEQ     NEWIDX              ;REPLACE INDEX IF MINIMUM = ELEMENT
        BCC     NXTBYT              ;BRANCH IF CURRENT MINIMUM < ELEMENT
                                    ;ELSE ELEMENT <= CURRENT MINIMUM
        LDA     (ARYADR),Y          ; NEW CURRENT MINIMUM AND
NEWIDX: STY     INDEX               ; NEW INDEX
NXTBYT:
        DEY                         ;DECREMENT TO NEXT BYTE
        BNE     MINLP

        ;
        ;EXIT
OKEXIT:
        LDY     INDEX               ;GET INDEX OF THE MINIMUM ELEMENT
        DEY                         ;NORMALIZE INDEX TO (0,SIZE-1)
        CLC                         ;NO ERRORS
        RTS

EREXIT:
        SEC                         ;ERROR, NO ELEMENTS IN THE ARRAY
        RTS

;DATA SECTION
INDEX:  .BLOCK  1                   ;INDEX OF SMALLEST ELEMENT

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC0904:
        LDA     AADR+1              ;A,Y = STARTING ADDRESS OF ARRAY
        LDY     AADR
        LDX     #SZARY              ;X = SIZE OF ARRAY
        JSR     MINELM
        BRK                         ;RESULT FOR THE INITIAL TEST DATA IS
                                    ; A = 01H (MINIMUM), Y=07 (INDEX TO MINIMUM)
        JMP     SC0904              ;LOOP FOR MORE TESTING


SZARY:  .EQU    10H                 ;SIZE OF ARRAY
AADR:   .WORD   ARY                 ;STARTING ADDRESS OF ARRAY
ARY:    .BYTE   8
        .BYTE   7
        .BYTE   6
        .BYTE   5
        .BYTE   4
```

```
.BYTE    3
.BYTE    2
.BYTE    1
.BYTE    0FFH
.BYTE    0FEH
.BYTE    0FDH
.BYTE    0FCH
.BYTE    0FBH
.BYTE    0FAH
.BYTE    0F9H
.BYTE    0F8H

.END     ;PROGRAM
```

Searches an array of unsigned byte-length elements for a particular value. The array is assumed to be ordered with the smallest element at the starting (lowest) address. Returns the index to the value and the Carry flag cleared if it finds the value; returns the Carry flag set to 1 if it does not find the value. The size of the array is specified and is a maximum of 255 bytes. The approach used is a binary search in which the value is compared with the middle element in the remaining part of the array; if the two are not equal, the part of the array that cannot possibly contain the value (because of the ordering) is discarded and the process is repeated.

Procedure: The program retains upper and lower bounds (indexes) that specify the part of the array still being searched. In each iteration, the new trial index is the average of the upper and lower bounds. The program compares the value and the element with the trial index; if the two are not equal, the program discards the part of the array that could not possibly contain the element. That is, if the value is larger than the element with the trial index, the part at or below the trial index is discarded. If the value is smaller than the element with the trial index, the part at or above the trial index is discarded. The program exits if it finds a match or if there are no elements left to be searched (that is, if the part of the array being searched no longer contains anything). The program sets the Carry flag to 1 if it finds the value and to 0 if it does not.

In the case of Example 1 shown later (the value is $0D_{16}$), the procedure works as follows:

In the first iteration, the lower bound is zero and the upper bound is the length of the array minus 1 (since we have started our indexing at zero). So we have

LOWER BOUND = 0
UPPER BOUND = LENGTH − 1 = $0F_{16} = 15_{10}$
GUESS = (UPPER BOUND + LOWER BOUND)/2 = 07 (the result is truncated)

ARRAY(GUESS) = ARRAY (7) = $10_{16} = 16_{10}$

Since our value ($0D_{16}$) is less than ARRAY(7), there is no use looking at the elements with indexes of 7 or more, so we have

LOWER BOUND = 0
UPPER BOUND = GUESS − 1 = 06

Registers Used: All

Execution Time: Approximately 52 cycles per iteration plus 80 cycles overhead. A binary search will require on the order of $\log_2 N$ iterations, where N is the size of the array (number of elements).

If, for example, N = 32, the binary search will require approximately $\log_2 32$ iterations or 5 iterations. The execution time will then be approximately

52 * 5 + 80 = 260 + 80 = 340 cycles.

Program Size: 89 bytes

Data Memory Required: Three bytes anywhere in RAM plus two bytes on page 0. The three bytes anywhere in RAM hold the value being searched for (one byte at address VALUE), the lower bound of the area being searched (one byte at address LBND), and the upper bound of the area being searched (one byte at address UBND). The two bytes on page 0 hold a pointer to the array (starting at address ARYADR, $00D0_{16}$ in the listing).

Special Case: A size or length of zero causes an immediate exit with the Carry flag set to 1. That is, the length is assumed to be zero and the value surely cannot be found.

GUESS = (UPPER BOUND + LOWER BOUND)/2 = 03
ARRAY(GUESS) = ARRAY(3) = 07

Since our value ($0D_{16}$) is greater than ARRAY (3), there is no use looking at the elements with indexes of 3 or less, so we have

LOWER BOUND = GUESS + 1 = 04
UPPER BOUND = 06
GUESS = (UPPER BOUND + LOWER BOUND)/2 = 05
ARRAY (GUESS) = ARRAY(5) = 09

Since our value ($0D_{16}$) is greater than ARRAY(5), there is no use looking at the elements with indexes of 5 or less, so we have

LOWER BOUND = GUESS + 1 = 06
UPPER BOUND = 06
GUESS = (UPPER BOUND + LOWER BOUND)/2 = 06
ARRAY(GUESS) = ARRAY(6) = $0D_{16}$

Since our value ($0D_{16}$) is equal to ARRAY(6), we have found the element. If, on the other hand, our value were $0E_{16}$, the new lower bound would be 07 and there would no longer be any elements in the part of the array left to be searched.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Value to find

Size of the array (in bytes)

Less significant byte of starting address of array (address of smallest unsigned element)
More significant byte of starting address of array (address of smallest unsigned element)

Exit Conditions

Carry = 0 if the value is found, Carry = 1 if it is not found. If the value is found, (A) = index to the value in the array.

Examples

Length of array = $10_{16} = 16_{10}$
Elements of array are 01_{16}, 02_{16}, 05_{16}, 07_{16}, 09_{16}, 09_{16}, $0D_{16}$, 10_{16}, $2E_{16}$, 37_{16}, $5D_{16}$, $7E_{16}$, $A1_{16}$, $B4_{16}$, $D7_{16}$, $E0_{16}$

1. Data: Value to find = $0D_{16}$
   Result: Carry = 0, indicating value found
   (A) = 06, the index of the value in the array

2. Data: Value to find = $9B_{16}$
   Result: Carry = 1, indicating value not found

```
;       Title               Binary Search                                   ;
;       Name:               BINSCH                                          ;
;                                                                           ;
;                                                                           ;


;                                                                           ;
;       Purpose:            Search an ordered array of unsigned bytes,      ;
;                           with a maximum size of 255 elements.            ;
;                                                                           ;
;       Entry:              TOP OF STACK                                    ;
;                             Low byte of return address,                   ;
;                             High byte of return address,                  ;
;                             Value to find,                                ;
;                             Length (size) of array,                       ;
;                             Low byte of starting array address,           ;
;                             High byte of starting array address           ;
;                                                                           ;
;       Exit:               If the value is found then                      ;
;                             CARRY flag = 0                                ;
;                             Register A = index to the value in the array  ;
;                           ELSE                                            ;
;                             CARRY flag = 1                                ;
;                                                                           ;
;       Registers used: All                                                 ;
;                                                                           ;
;       Time:               Approximately 52 cycles for each time through   ;
;                           the search loop plus 80 cycles overhead.        ;
;                           A binary search will take on the order of log   ;
;                           base 2 of N searches, where N is the number of  ;
;                           elements in the array.                          ;
;                                                                           ;
;       Size:               Program 89 bytes                                ;
;                           Data     3 bytes plus                           ;
;                                    2 bytes in page zero                   ;
;                                                                           ;
;                                                                           ;


;
;EQUATES SECTION
ARYADR: .EQU    0D0H    ;PAGE ZERO POINTER TO ARRAY

;
BINSCH:
        ;GET RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET THE VALUE TO SEARCH FOR
        PLA
        STA     VALUE

        ;GET THE LENGTH OF THE ARRAY
        PLA
        STA     UBND
```

```
        ;GET THE STARTING ADDRESS OF ARRAY
        PLA
        STA     ARYADR
        PLA
        STA     ARYADR+1

        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA

        ;
        ;CHECK THAT LENGTH IS NOT ZERO
        LDX     UBND            ;GET LENGTH
        BEQ     NOTFND          ;EXIT NOT FOUND IF LENGTH EQUALS ZERO

        ;SET UPPER AND LOWER SEARCH BOUNDS
        DEX
        STX     UBND            ;UPPER BOUND EQUALS LENGTH MINUS 1
        LDA     #0
        STA     LBND            ;LOWER BOUND EQUALS 0

        ;SEARCH LOOP
        ;  COMPUTE NEXT INDEX TO BE HALF WAY BETWEEN UPPER BOUND AND
        ;  LOWER BOUND
NXTBYT:
        LDA     UBND
        CLC
        ADC     LBND            ;ADD LOWER AND UPPER BOUNDS
        ROR     A               ;DIVIDE BY 2, TRUNCATING FRACTION
        TAY                     ;REGISTER Y BECOMES INDEX

        ;IF INDEX IS GREATER THAN UPPER BOUND THEN THE ELEMENT IS NOT HERE
        CPY     UBND
        BEQ     TSTLB           ;BRANCH IF INDEX EQUALS UPPER BOUND
        BCS     NOTFND          ;BRANCH IF INDEX IS GREATER THAN UPPER BOUND

        ;IF INDEX IS LESS THAN LOWER BOUND THEN THE ELEMENT IS NOT HERE
TSTLB:
        CPY     LBND
        BCC     NOTFND          ;BRANCH IF INDEX IS LESS THAN LOWER BOUND

        ;TEST IF WE HAVE FOUND THE ELEMENT
        LDA     VALUE
        CMP     (ARYADR),Y
        BCC     SMALL           ;BRANCH IF VALUE IS SMALLER THAN ARYADR[Y]
        BEQ     FND             ;BRANCH IF FOUND

        ;VALUE IS LARGER THAN ARYADR[Y] SO SET LOWER BOUND TO BE
        ; Y + 1 (VALUE CAN ONLY BE FURTHER UP)
        INY
        STY     LBND
        BNE     NXTBYT          ;CONTINUE SEARCHING IF LOWER BOUND DOES NOT
                                ; OVERFLOW
        BEQ     NOTFND          ;BRANCH IF LOWER BOUND OVERFLOWED FROM 0FFH
                                ; TO 0
```

```
        ;VALUE IS SMALLER THAN ARYADR[Y] SO SET UPPER BOUND TO BE
        ; Y - 1 (VALUE CAN ONLY BE FURTHER DOWN)
SMALL:
        DEY
        STY     UBND
        CPY     #0FFH
        BNE     NXTBYT          ;CONTINUE SEARCHING IF UPPER BOUND DOES NOT
                                ; UNDERFLOW
        BEQ     NOTFND          ;BRANCH IF INDEX UNDERFLOWED

        ;FOUND THE VALUE
FND:
        CLC                     ;INDICATE VALUE FOUND
        TYA                     ;GET INDEX OF VALUE TO REGISTER A
        RTS

        ;DID NOT FIND THE VALUE
NOTFND:
        SEC                     ;INDICATE VALUE NOT FOUND
        RTS

;DATA SECTION
VALUE   .BLOCK  1               ;VALUE TO FIND
LBND    .BLOCK  1               ;INDEX OF LOWER BOUND
UBND    .BLOCK  1               ;INDEX OF UPPER BOUND


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;


;
SC0905:
        ;SEARCH FOR A VALUE WHICH IS IN THE ARRAY
        LDA     BFADR+1
        PHA                     ;PUSH HIGH BYTE OF STARTING ADDRESS
        LDA     BFADR
        PHA                     ;PUSH LOW BYTE OF STARTING ADDRESS
        LDA     BFSZ
        PHA                     ;PUSH LENGTH (SIZE OF ARRAY)
        LDA     #7
        PHA                     ;PUSH VALUE TO FIND
        JSR     BINSCH          ;SEARCH
        BRK                     ;CARRY FLAG SHOULD BE 0 AND REGISTER A = 4

        ;SEARCH FOR A VALUE WHICH IS NOT IN THE ARRAY
        LDA     BFADR+1
        PHA                     ;PUSH HIGH BYTE OF STARTING ADDRESS
        LDA     BFADR
        PHA                     ;PUSH LOW BYTE OF STARTING ADDRESS
        LDA     BFSZ
        PHA                     ;PUSH LENGTH (SIZE OF ARRAY)
        LDA     #0
        PHA                     ;PUSH VALUE TO FIND
```

```
        JSR     BINSCH          ;SEARCH
        BRK                     ;CARRY FLAG SHOULD BE 1

        JMP     SC0905          ;LOOP FOR MORE TESTS
;
;DATA
SIZE    .EQU    010H            ;SIZE OF BUFFER
BFADR:  .WORD   BF              ;STARTING ADDRESS OF BUFFER
BFSZ:   .BYTE   SIZE            ;SIZE OF BUFFER
BF:                             ;BUFFER
        .BYTE   1
        .BYTE   2
        .BYTE   4
        .BYTE   5
        .BYTE   7
        .BYTE   9
        .BYTE   10
        .BYTE   11
        .BYTE   23
        .BYTE   50
        .BYTE   81
        .BYTE   123
        .BYTE   191
        .BYTE   199
        .BYTE   250
        .BYTE   255

        .END    ;PROGRAM
```

Bubble Sort (BUBSRT)

Arranges an array of unsigned byte-length elements into ascending order using a bubble sort algorithm. An iteration of this algorithm moves the largest remaining element to the top by comparisons with all other elements, performing interchanges if necessary along the way. The algorithm continues until it has either worked its way through all elements or has completed an iteration without interchanging anything. The size of the array is specified and is a maximum of 255 bytes.

Procedure: The program starts by considering the entire array. It examines pairs of elements, interchanging them if they are out of order and setting a flag to indicate that the interchange occurred. At the end of an iteration, the program checks the interchange flag to see if the array is already in order. If it is not, the program performs another iteration, reducing the number of elements examined by one since the largest remaining element has been bubbled to the top. The program exits immediately if the length of the array is less than two, since no ordering is then necessary. Note that the number of pairs is always one less than the number of elements being considered, since the last element has no successor.

Registers Used: All

Execution Time: Approximately

$34 * N * N + 25 * N + 70$

cycles, where N is the size (length) of the array in bytes. If, for example, N is 20_{16} (32_{10}), the execution time is approximately

$34 * 32 * 32 + 25 * 32 + 70 = 34 * 1024 + 870$
$= 34{,}816 + 870 = 35{,}686$ cycles.

Program Size: 79 bytes

Data Memory Required: Two bytes anywhere in RAM plus four bytes on page 0. The two bytes anywhere in RAM hold the length of the array (one byte at address LEN) and the interchange flag (one byte at address XCHGFG). The four bytes on page 0 hold pointers to the first and second elements of the array (two bytes starting at address A1ADR, $00D0_{16}$ in the listing, and two bytes starting at address A2ADR, $00D2_{16}$ in the listing).

Special Case: A size (or length) of 00 or 01 causes an immediate exit with no sorting.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Length (size) of array in bytes

Less significant byte of starting address of array

More significant byte of starting address of array

Exit Conditions

Array sorted into ascending order, considered the elements as unsigned bytes. Thus, the smallest unsigned byte is now in the starting address.

Example

Data: Length (size) of array = 06
Elements = 35_{16}, $6A_{16}$, $2B_{16}$, $3E_{16}$, $D4_{16}$, $4F_{16}$

Result: After the first iteration, we have
35_{16}, $2B_{16}$, $3E_{16}$, $6A_{16}$, $4F_{16}$, $D4_{16}$.
The largest element is now at the end of the array and need not be considered further.

After the second iteration, we have
$2B_{16}$, 35_{16}, $3E_{16}$, $4F_{16}$, $6A_{16}$, $D4_{16}$.
The next to largest element is now in the correct position and need not be considered further.

The third iteration leaves the array unchanged, since the elements are already in ascending order.

```
;       Title                 Bubble sort                                  ;
;       Name:                 BUBSRT                                       ;
;                                                                          ;
;                                                                          ;
;                                                                          ;
;       Purpose:              Arrange an array of unsigned bytes into      ;
;                             ascending order using a bubble sort, with a  ;
;                             maximum size of 255 bytes.                   ;
;                                                                          ;
;       Entry:                TOP OF STACK                                 ;
;                               Low byte of return address,                ;
;                               High byte of return address,               ;
;                               Length (size) of array,                    ;
;                               Low byte of starting array address,        ;
;                               High byte of starting array address        ;
;                                                                          ;
;       Exit:                 The array is sorted into ascending order.    ;
;                                                                          ;
;       Registers used: All                                                ;
;                                                                          ;
;       Time:                 Approximately (34 * N * N) + (25 * N) cycles ;
;                             plus 70 cycles overhead, where N is the size of ;
;                             the array.                                   ;
;                                                                          ;
;       Size:                 Program 79 bytes                             ;
;                             Data     2 bytes plus                        ;
;                                      4 bytes in page zero                ;
;                                                                          ;
;                                                                          ;

;
;EQUATES SECTION
A1ADR:  .EQU    0D0H            ;ADDRESS OF FIRST ELEMENT
A2ADR:  .EQU    0D2H            ;ADDRESS OF SECOND ELEMENT

;
BUBSRT:
        ;GET THE PARAMETERS FROM THE STACK
        PLA
        TAY                     ;SAVE LOW BYTE OF RETURN ADDRESS
```

```
        PLA
        TAX                             ;SAVE HIGH BYTE OF RETURN ADDRESS

        PLA
        STA     LEN                     ;SAVE THE LENGTH (SIZE)
        PLA
        STA     A1ADR                   ;SAVE THE LOW BYTE OF THE ARRAY ADDRESS
        CLC
        ADC     #1
        STA     A2ADR                   ;SET LOW BYTE OF A2ADR TO A1ADR + 1
        PLA
        STA     A1ADR+1                 ;SAVE THE HIGH BYTE OF THE ARRAY ADDRESS
        ADC     #0
        STA     A2ADR+1                 ;SET HIGH BYTE OF A2ADR
        TXA
        PHA                             ;RESTORE HIGH BYTE OF RETURN ADDRESS
        TYA
        PHA                             ;RESTORE LOW BYTE OF RETURN ADDRESS


        ;BE SURE THE LENGTH IS GREATER THAN 1
        LDA     LEN
        CMP     #2
        BCC     DONE                    ;EXIT IF THE LENGTH OF THE ARRAY IS
                                        ; LESS THAN 2

        ;REDUCE LENGTH BY 1 SINCE THE LAST ELEMENT HAS NO SUCCESSOR
        DEC     LEN

        ;BUBBLE SORT LOOP
SRTLP:
        LDX     LEN                     ;X BECOMES NUMBER OF TIMES THROUGH INNER LOOP
        LDY     #0                      ;Y BECOMES BEGINNING INDEX
        STY     XCHGFG                  ;INITIALIZE EXCHANGE FLAG TO 0
INLOOP:
        LDA     (A2ADR),Y
        CMP     (A1ADR),Y               ;COMPARE 2 ELEMENTS
        BCS     AFTSWP                  ;BRANCH IF SECOND ELEMENT >= FIRST ELEMENT
        PHA                             ;SECOND ELEMENT LESS, SO EXHANGE ELEMENTS
        LDA     (A1ADR),Y               ;GET SECOND ELEMENT
        STA     (A2ADR),Y               ;STORE IT INTO THE FIRST ELEMENT
        PLA
        STA     (A1ADR),Y               ;STORE FIRST ELEMENT INTO SECOND
        LDA     #1
        STA     XCHGFG                  ;SET EXCHANGE FLAG SINCE AN EXCHANGE OCCURRED

AFTSWP:
        INY                             ;INCREMENT TO NEXT ELEMENT
        DEX
        BNE     INLOOP                  ;BRANCH NOT DONE WITH INNER LOOP

        ;INNER LOOP IS COMPLETE IF THERE WERE NO EXCHANGES THEN EXIT
        LDA     XCHGFG                  ;GET EXCHANGE FLAG
        BEQ     DONE                    ;EXIT IF NO EXCHANGE WAS PERFORMED
        DEC     LEN
        BNE     SRTLP                   ;CONTINUE IF LENGTH IS NOT ZERO
```

```
DONE:
        RTS

;DATA SECTION
LEN:    .BLOCK  1                        ;LENGTH OF THE ARRAY
XCHGFG: .BLOCK  1                        ;EXCHANGE FLAG (1=EXCHANGE, 0=NO EXCHANGE)

;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;


;
;PROGRAM SECTION
SC0906:
        ;SORT AN ARRAY
        LDA     BFADR+1
        PHA                              ;PUSH HIGH BYTE OF STARTING ADDRESS
        LDA     BFADR
        PHA                              ;PUSH LOW BYTE OF STARTING ADDRESS
        LDA     BFSZ
        PHA                              ;PUSH LENGTH (SIZE OF ARRAY)
        JSR     BUBSRT                   ;SORT
        BRK                              ;THE RESULT FOR THE INITIAL TEST DATA IS
                                         ; 0,1,2,3, ... ,14,15
        JMP     SC0906                   ;LOOP FOR MORE TESTS

;
;DATA SECTION
SIZE    .EQU    010H                     ;SIZE OF BUFFER
BFADR:  .WORD   BF                       ;STARTING ADDRESS OF BUFFER
BFSZ:   .BYTE   SIZE                     ;SIZE OF BUFFER
BF:                                      ;BUFFER
        .BYTE   15
        .BYTE   14
        .BYTE   13
        .BYTE   12
        .BYTE   11
        .BYTE   10
        .BYTE   9
        .BYTE   8
        .BYTE   7
        .BYTE   6
        .BYTE   5
        .BYTE   4
        .BYTE   3
        .BYTE   2
        .BYTE   1
        .BYTE   0

        .END    ;PROGRAM
```

Performs a test of an area of RAM memory specified by a starting address and a length in bytes. Writes the values 00, FF_{16}, AA_{16} (10101010_2), and 55_{16} (01010101_2) into each byte and checks to see if they can be read back correctly. Places a single 1 bit in each position of each byte and sees if that can be read back correctly. Clears the Carry flag if all tests can be performed; if it finds an error it immediately exits, setting the Carry flag and returning the address in which the error occurred and the value that was being used in the test.

Procedure: The program performs the single value checks (with 00, FF_{16}, AA_{16}, and 55_{16}) by first filling the memory area and then comparing each byte with the specified value. Filling the entire area first should provide enough delay between writing and reading to detect a failure to retain data (perhaps caused by improperly designed refresh circuitry). The program then performs the walking bit test, starting with bit 7; here it writes the data into memory and immediately attempts to read it back for a comparison. In all the tests, the program handles complete pages first and then handles the remaining partial page; the program can thus use 8-bit counters rather than a 16-bit counter. This approach reduces execution time but increases memory usage as compared to handling the entire area with one loop. Note that the program exits immediately if it finds an error, setting the Carry flag to 1 and returning the location and the value being used in the test. If all the tests can be performed correctly, the program clears the Carry flag before exiting.

Registers Used: All

Execution Time: Approximately 245 cycles per byte tested plus 650 cycles overhead. Thus, for example, to test an area of size $0400_{16} = 1024_{10}$ would take

$$245 * 1024 + 650 = 250{,}880 + 650 = 251{,}530 \text{ cycles.}$$

Program Size: 229 bytes

Data Memory Required: Six bytes anywhere in RAM plus two bytes on page 0. The six bytes anywhere in RAM hold the address of the first element (two bytes starting at address ADDR), the length of the tested area (two bytes starting at address LEN), and the temporary length (two bytes starting at address TLEN). The two bytes on page 0 hold a pointer to the tested area (starting at address TADDR, $00D0_{16}$ in the listing).

Special Cases:

1. An area size of 0000_{16} causes an immediate exit with no memory tested. The Carry flag is cleared to indicate no errors.

2. Since the routine changes all bytes in the tested area, using it to test an area that includes its own temporary storage will produce unpredictable results.

Note that Case 1 means you cannot ask this routine to test the entire memory, but such a request would be meaningless anyway since it would require the routine to test its own temporary storage.

3. Attempting to test a ROM area will cause a return with an error indication as soon as the program attempts to store a value in a ROM location that is not already there.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address
Less significant byte of size (length) of area in bytes
More significant byte of size (length) of area in bytes
Less significant byte of starting address of test area
More significant byte of starting address of test area

Exit Conditions

1. If an error is found,
   Carry = 1
   (A) = More significant byte of address containing error
   (Y) = Less significant byte of address containing error
   (X) = Expected value (value being used in test)
2. If no error is found,
   Carry = 0
   All bytes in test area contain 00.

Example

Data: Starting address = 0380_{16}
Length (size) of area = 0200_{16}

Result: Area tested is the 0200_{16} bytes, starting at addresses 0380_{16}. That is, address 0380_{16} through $057F_{16}$. The order of the tests is:

1. Write and read 00
2. Write and read FF_{16}
3. Write and read AA_{16} (10101010_2)
4. Write and read 55_{16} (01010101_2)
5. Walking bit test, starting with bit 7 and moving right. That is, starting with 80_{16} (10000000_2) and moving the 1 bit one position right in each subsequent test of a single byte.

```
;       Title              RAM test                                 ;
;       Name:              RAMTST                                   ;
;                                                                   ;
;                                                                   ;

;                                                                   ;
;       Purpose:           Perform a test of RAM memory             ;
;                            1) Write all 00 hex and test           ;
;                            2) Write all FF hex and test           ;
;                            3) Write all AA hex and test           ;
;                            4) Write all 55 hex and test           ;
;                            5) Shift a single 1 bit thourgh all of memory ;
;                                                                   ;
;                            If the program finds an error, it exits ;
;                            immediately with the CARRY flag set and ;
;                            indicates where the error occurred and ;
;                            what value it used in the test.        ;
;                                                                   ;
```

```
;       Entry:             TOP OF STACK                                   ;
;                            Low byte of return address,                  ;
;                            High byte of return address,                 ;
;                            Low byte of length in bytes,                 ;
;                            High byte of length in bytes,                ;
;                            Low byte of starting address of test area,   ;
;                            High byte of starting address of test area   ;
;                                                                         ;
;       Exit:              If there are no errors then                    ;
;                            CARRY flag equals 0                          ;
;                            test area contains 00 in all bytes           ;
;                          else                                           ;
;                            CARRY flag equals 1                          ;
;                            Register A = High byte of the address        ;
;                                         containing the error            ;
;                            Register Y = Low byte of the address         ;
;                                         containing the error            ;
;                            Register X = Expected value                  ;
;                                                                         ;
;       Registers used: All                                               ;
;                                                                         ;
;       Time:              Approximately 245 cycles per byte plus         ;
;                          650 cycles overhead.                           ;
;                                                                         ;
;       Size:              Program 228 bytes                              ;
;                          Data      6 bytes plus                         ;
;                                    2 bytes in page zero                 ;
;                                                                         ;
;                                                                         ;

;
;EQUATES SECTION

TADDR:  .EQU    0D0H            ;PAGE ZERO POINTER TO TEST AREA

;
RAMTST:
        ;GET THE RETURN ADDRESS
        PLA
        TAY
        PLA
        TAX

        ;GET THE LENGTH OF THE TEST AREA
        PLA
        STA     LEN
        PLA
        STA     LEN+1

        ;GET THE STARTING ADDRESS OF THE TEST AREA
        PLA
        STA     ADDR
        PLA
        STA     ADDR+1
```

```
        ;RESTORE THE RETURN ADDRESS
        TXA
        PHA
        TYA
        PHA

        ;BE SURE THE LENGTH IS NOT ZERO
        LDA     LEN
        ORA     LEN+1
        BEQ     EXITOK          ;EXIT WITH NO ERRORS IF LENGTH IS ZERO

        ;FILL MEMORY WITH FF HEX (ALL 1'S) AND COMPARE
        LDA     #0FFH
        JSR     FILCMP
        BCS     EXITER          ;EXIT IF AN ERROR

        ;FILL MEMORY WITH AA HEX (ALTERNATING 1'S AND 0'S) AND COMPARE
        LDA     #0AAH
        JSR     FILCMP
        BCS     EXITER          ;EXIT IF AN ERROR

        ;FILL MEMORY WITH 55 HEX (ALTERNATING 0'S AND 1'S) AND COMPARE
        LDA     #55H
        JSR     FILCMP
        BCS     EXITER          ;EXIT IF AN ERROR

        ;FILL MEMORY WITH 0 AND COMPARE
        LDA     #0
        JSR     FILCMP
        BCS     EXITER

        ;PERFORM WALKING BIT TEST
        JSR     ITEMPS          ;INITIALIZE TEMPORARIES

        ;WALK THROUGH THE 256 BYTE PAGES
        LDX     TLEN+1          ;CHECK IF ANY FULL PAGES TO DO
        BEQ     WLKPRT          ;BRANCH IF NONE
        LDY     #0              ;REGISTER Y IS INDEX

WLKLP:
        LDA     #80H            ;SET BIT 7 TO 1, ALL OTHER BITS TO ZERO
WLKLP1:
        STA     (TADDR),Y       ;STORE TEST PATTERN IN MEMORY
        CMP     (TADDR),Y       ;COMPARE VALUE WITH WHAT IS READ BACK
        BNE     EXITER          ;EXIT INDICATING ERROR IF NOT THE SAME
        LSR     A               ;SHIFT TEST PATTERN RIGHT ONE BIT
        BNE     WLKLP1          ;BRANCH IF NOT DONE WITH BYTE
        STA     (TADDR),Y       ;STORE A ZERO BACK INTO THE LAST BYTE
        INY                     ;INCREMENT TO NEXT BYTE IN PAGE
        BNE     WLKLP           ;BRANCH IF NOT DONE WITH PAGE
        INC     TADDR+1         ;INCREMENT TO NEXT PAGE
        DEX                     ;DECREMENT PAGE COUNTER
        BNE     WLKLP           ;BRANCH IF NOT DONE WITH ALL OF THE PAGES

        ;WALK THROUGH LAST PARTIAL PAGE
```

```
WLKPRT:
        LDX     TLEN            ;GET NUMBER OF BYTES IN LAST PAGE
        BEQ     EXITOK          ;EXIT IF NONE
        LDY     #0              ;INITIALIZE INDEX TO ZERO

WLKLP2:
        LDA     #80H            ;START WITH BIT 7 EQUAL TO 1
WLKLP3:
        STA     (TADDR),Y       ;STORE TEST PATTERN IN MEMORY
        CMP     (TADDR),Y       ;COMPARE VALUE WITH WHAT IS READ BACK
        BNE     EXITER          ;EXIT INDICATING ERROR IF NOT THE SAME
        LSR     A               ;SHIFT TEST PATTERN RIGHT
        BNE     WLKLP3          ;BRANCH IF NOT DONE
        STA     (TADDR),Y       ;STORE A ZERO BACK INTO THE LAST BYTE
        INY                     ;INCREMENT TO NEXT BYTE
        DEX                     ;DECREMENT BYTE COUNTER
        BNE     WLKLP2          ;BRANCH IF NOT DONE

EXITOK:
        CLC                     ;RETURN WITH NO ERROR
        RTS

EXITER:
        JSR     ERROR           ;RETURN WITH AN ERROR
        RTS


;************************************
;ROUTINE: FILCMP
;PURPOSE: FILL MEMORY WITH A VALUE AND TEST
;         THAT MEMORY CONTAINS THAT VALUE
;ENTRY: REGISTER A = VALUE
;       ADDR = STARTING ADDRESS
;       LEN = LENGTH
;EXIT:  IF NO ERRORS THEN
;         CARRY FLAG EQUALS 0
;       ELSE
;         CARRY FLAG EQUALS 1
;         REGISTER A = HIGH BYTE OF ERROR LOCATION
;         REGISTER Y = LOW BYTE OF ERROR LOCATION
;         REGISTER X = EXPECTED VALUE
;REGISTERS USED: ALL
;************************************

FILCMP:
        JSR     ITEMPS          ;INITIALIZE TEMPORARIES

        ;FILL MEMORY WITH THE VALUE IN REGISTER A
        ;FILL FULL PAGES
        LDX     TLEN+1
        BEQ     FILPRT
        LDY     #0              ;START AT INDEX 0
FILLP:
        STA     (TADDR),Y       ;STORE THE VALUE
        INY                     ;INCREMENT TO NEXT LOCATION
        BNE     FILLP           ;BRANCH IF NOT DONE WITH THIS PAGE
```

```
        INC     TADDR+1         ;INCREMENT HIGH BYTE OF TEMPORARY ADDRESS
        DEX                     ;DECREMENT PAGE COUNT
        BNE     FILLP           ;BRANCH IF NOT DONE WITH FILL

FILPRT:
        ;FILL PARTIAL PAGE
        LDX     TLEN            ;REGISTER Y IS SET TO SIZE OF LAST PAGE
        LDY     #0
FILLP1:
        STA     (TADDR),Y
        INY
        DEX
        BNE     FILLP1          ;CONTINUE


        ;COMPARE MEMORY AGAINST THE VALUE IN REGISTER A
CMPARE:
        JSR     ITEMPS          ;INITIALIZE TEMPORARIES

        ;COMPARE MEMORY WITH THE VALUE IN REGISTER A
        ; COMPARE FULL PAGES FIRST
        LDX     TLEN+1
        BEQ     CMPPRT
        LDY     #0              ;START AT INDEX 0
CMPLP:
        CMP     (TADDR),Y       ;CAN THE STORED VALUE BE READ BACK ?
        BNE     CMPER           ;NO, EXIT INDICATING ERROR
        INY                     ;INCREMENT TO NEXT LOCATION
        BNE     CMPLP           ;BRANCH IF NOT DONE WITH THIS PAGE
        INC     TADDR+1         ;INCREMENT HIGH BYTE OF TEMPORARY ADDRESS
        DEX                     ;DECREMENT PAGE COUNT
        BNE     CMPLP           ;BRANCH IF NOT DONE WITH FILL

CMPPRT:
        ;COMPARE THE LAST PARTIAL PAGE
        LDX     TLEN            ;REGISTER Y = SIZE OF PARTIAL PAGE
        LDY     #0
CMPLP1:
        CMP     (TADDR),Y       ;CAN THE STORED VALUE BE READ BACK ?
        BNE     CMPER           ;NO, EXIT INDICATING ERROR
        INY
        DEX
        BNE     CMPLP1          ;CONTINUE


CMPOK:
        CLC                     ;INDICATE NO ERROR
        RTS

CMPER:
        JSR     ERROR
        RTS

;*************************************
;ROUTINE: ITEMPS
;PURPOSE: INITIALIZE TEMPORARIES
```

```
;ENTRY: ADDR IS BEGINNING ADDRESS
;       LEN IS NUMBER OF BYTES
;EXIT:  TADDR IS SET TO ADDR
;       TLEN IS SET TO LEN
;REGISTERS USED: Y,P
;***********************************

ITEMPS:
        LDY     ADDR
        STY     TADDR
        LDY     ADDR+1
        STY     TADDR+1

        LDY     LEN
        STY     TLEN
        LDY     LEN+1
        STY     TLEN+1
        RTS


;****************************************
;ROUTINE: ERROR
;PURPOSE: SET UP THE REGISTERS FOR AN ERROR EXIT
;ENTRY: REGISTER A IS EXPECTED BYTE
;       TADDR IS BASE ADDRESS
;       REGISTER Y IS INDEX
;EXIT   REGISTER X IS SET TO EXPECTED BYTE
;       REGISTER A IS SET TO HIGH BYTE OF THE ADDRESS CONTAINING THE ERROR
;       REGISTER Y IS SET TO LOW BYTE OF THE ADDRESS CONTAINING THE ERROR
;       CARRY FLAG IS SET TO 1
;REGISTERS USED: ALL
;***************************************

ERROR:
        TAX                     ;REGISTER X = EXPECTED BYTE
        TYA                     ;GET INDEX
        CLC                     ;ADDRESS OF ERROR = BASE + INDEX
        ADC     TADDR
        TAY                     ;REGISTER Y = LOW BYTE OF ERROR LOCATION
        LDA     TADDR+1
        ADC     #0              ;REGISTER A = HIGH BYTE OF ERROR LOCATION
        SEC                     ;INDICATE AN ERROR BY SETTING CARRY TO 1
        RTS


;DATA SECTION
ADDR:   .BLOCK  2               ;ADDRESS OF FIRST ELEMENT
LEN:    .BLOCK  2               ;LENGTH
TLEN:   .BLOCK  2               ;TEMPORARY LENGTH

;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION                                        ;
;                                                               ;
;                                                               ;
```

```
SC0907:
        ;TEST MEMORY
        LDA     ADR+1
        PHA                     ;PUSH HIGH BYTE OF STARTING ADDRESS
        LDA     ADR
        PHA                     ;PUSH LOW BYTE OF STARTING ADDRESS
        LDA     SZ+1
        PHA                     ;PUSH HIGH BYTE OF LENGTH
        LDA     SZ
        PHA                     ;PUSH LOW BYTE OF LENGTH
        JSR     RAMTST          ;TEST
        BRK                     ;CARRY FLAG SHOULD BE 0
        JMP     SC0907          ;LOOP FOR MORE TESTING

ADR     .WORD   2000H
SZ      .WORD   1010H

        .END    ;PROGRAM
```

Jump Table (JTAB)

Transfers control to an address selected from a table according to an index. The addresses are stored in the usual 6502 style (less significant byte first), starting at address TABLE. The size of the table (number of addresses) is a constant LENSUB, which must be less than or equal to 128. If the index is greater than or equal to LENSUB, the program returns control immediately with the Carry flag set to 1.

Procedure: The program first checks if the index is greater than or equal to the size of the table (LENSUB). If it is, the program returns control with the Carry flag set. If it is not, the program obtains the starting address of the appropriate subroutine from the table, stores it in memory, and jumps to it indirectly.

Registers Used: A, P

Execution Time: 31 cycles overhead, besides the time required to execute the subroutine.

Program Size: 23 bytes plus 2*LENSUB bytes for the table of starting addresses, where LENSUB is the number of subroutines.

Data Memory Required: Two bytes anywhere in RAM (starting at address TMP) to hold the indirect address obtained from the table.

Special Case: Entry with (A) greater than or equal to LENSUB causes an immediate exit with Carry flag set to 1.

Entry Conditions

(A) = index

Exit Conditions

If (A) is greater than LENSUB, an immediate return with Carry = 1. Otherwise, control transferred to appropriate subroutine as if an indexed call had been performed. The return address remains at the top of the stack.

Example

Data: LENSUB (size of subroutine table) = 03.
Table consists of addresses SUB0, SUB1, and SUB2.
Index = (A) = 02

Result: Control transferred to address SUB2 (PC = SUB2).

```
;        Title                Jump table                                      ;
;        Name:                JTAB                                            ;
;                                                                             ;
;                                                                             ;
;                                                                             ;
;                                                                             ;
;        Purpose:             Given an index, jump to the subroutine with     ;
;                             that index in a table                           ;
;                                                                             ;
;        Entry:               Register A is the subroutine number 0 to        ;
;                                       LENSUB-1, the number of subroutines,  ;
;                                       LENSUB must be less than or equal to  ;
;                                       128.                                  ;
;                                                                             ;
;        Exit:                If the routine number is valid then             ;
;                               execute the routine                           ;
;                             else                                            ;
;                               CARRY flag equals 1                           ;
;                                                                             ;
;        Registers used: A,P                                                  ;
;                                                                             ;
;        Time:                31 cycles plus execution time of subroutine     ;
;                                                                             ;
;        Size:                Program 23 bytes plus size of table (2*LENSUB)  ;
;                             Data     2 bytes                                ;
;                                                                             ;
;                                                                             ;
;

;
JTAB:
        CMP     #LENSUB
        BCS     JTABER          ;BRANCH IF REGISTER A IS TOO LARGE
        ASL     A               ;MULTIPLY VALUE BY 2 FOR WORD-LENGTH INDEX
        TAY
        LDA     TABLE,Y         ;MOVE STARTING ADDRESS TO TEMPORARY STORAGE
        STA     TMP
        LDA     TABLE+1,Y
        STA     TMP+1
        JMP     (TMP)           ;JUMP INDIRECTLY TO SUBROUTINE

JTABER:
        SEC                     ;INDICATE A BAD ROUTINE NUMBER
        RTS

LENSUB  .EQU    3
TABLE:
        .WORD   SUB1            ;ROUTINE 0
        .WORD   SUB2            ;ROUTINE 1
        .WORD   SUB3            ;ROUTINE 2

TMP:    .BLOCK  2               ;TEMPORARY ADDRESS TO JUMP INDIRECT THROUGH
```

```
;THREE SUBROUTINES WHICH ARE IN THE JUMP TABLE
SUB1:
        LDA     #1
        RTS

SUB2:
        LDA     #2
        RTS

SUB3:
        LDA     #3
        RTS

;                                                        ;
;                                                        ;
;       SAMPLE EXECUTION                                 ;
;                                                        ;
;                                                        ;

;
;PROGRAM SECTION
SC0908:
        LDA     #0
        JSR     JTAB
        BRK                     ;EXECUTE ROUTINE 0, REGISTER A EQUALS 1
        LDA     #1
        JSR     JTAB
        BRK                     ;EXECUTE ROUTINE 1, REGISTER A EQUALS 2
        LDA     #2
        JSR     JTAB
        BRK                     ;EXECUTE ROUTINE 2, REGISTER A EQUALS 3
        LDA     #3
        JSR     JTAB
        BRK                     ;ERROR CARRY FLAG EQUALS 1
        JMP     SC0908          ;LOOP FOR MORE TESTS

        .END    ;PROGRAM
```

Read a Line of Characters from a Terminal (RDLINE)

10A

Reads ASCII characters from a terminal and saves them in a buffer until it encounters a carriage return character. Defines the control characters Control H (08 hex), which deletes the character most recently entered into the buffer, and Control X (18 hex), which deletes all characters in the buffer. Sends a bell character (07 hex) to the terminal if the buffer becomes full. Echoes to the terminal each character placed in the buffer. Sends a new line sequence (typically carriage return, line feed) to the terminal before exiting.

RDLINE assumes the existence of the following system-dependent subroutines:

1. RDCHAR reads a single character from the terminal and places it in the accumulator.
2. WRCHAR sends the character in the accumulator to the terminal.
3. WRNEWL sends a new line sequence (typically consisting of carriage return and line feed characters) to the terminal.

These subroutines are assumed to change the contents of all the user registers.

RDLINE is intended as an example of a typical terminal input handler. The specific control characters and I/O subroutines in a real system will, of course, be computer-dependent. A specific example in the listing describes an Apple II computer with the following features:

1. The entry point for the routine that reads a character from the keyboard is $FD0C_{16}$. This routine returns with bit 7 set, so that bit must be cleared for normal ASCII operations.
2. The entry point for the routine that sends a character to the monitor is $FDED_{16}$. This routine requires bit 7 of the character (in the accumulator) to be set.
3. The entry point for the routine that issues the appropriate new line character (a carriage return) is $FD8E_{16}$.
4. An 08_{16} character moves the cursor left one position.

A standard reference describing the Apple II computer is L. Poole et al., *Apple II User's Guide,* Berkeley: Osborne/McGraw-Hill, 1981.

Registers Used: All

Execution Time: Approximately 67 cycles to place an ordinary character in the buffer, not considering the execution time of either RDCHAR or WRCHAR.

Program Size: 138 bytes

Data Memory Required: Four bytes anywhere in RAM plus two bytes on page 0. The four bytes anywhere in RAM hold the buffer index (one byte at address BUFIDX), the buffer length (one byte at address BUFLEN), the count for the backspace routine (one byte at address COUNT), and the index for the backspace routine (one byte at address INDEX). The two bytes on page 0 hold a pointer to the input buffer (starting at address BUFADR, $00D0_{16}$ in the listing).

Special Cases:

1. Typing Control H (delete one character) or Control X (delete the entire line) when there is nothing in the buffer has no effect on the buffer and does not cause anything to be sent to the terminal.
2. If the program receives an ordinary character when the buffer is full, it sends a Bell character to the terminal (ringing the bell), discards the received character, and continues its normal operations.

Procedure: The program first reads a character using the RDCHAR routine and exits if the character is a carriage return. If the character is not a carriage return, the program checks for the special characters Control H and Control X. In response to Control H, the program decrements the buffer index and sends a backspace string (consisting of cursor left, space, cursor left) to the terminal if there is anything in the buffer. In response to Control X, the program repeats the response to Control H until it empties the buffer. If the character is not special, the program checks to see if the buffer is full. If the buffer is full, the program sends a bell character to the terminal and continues. If the buffer is not full, the program stores the character in the buffer, echoes it to the terminal, and adds one to the buffer index. Before exiting, the program sends a new line sequence to the terminal using the WRNEWL routine.

Entry Conditions

(A) = More significant byte of starting address of buffer

(Y) = Less significant byte of starting address of buffer

(X) = Length (size) of the buffer in bytes.

Exit Conditions

(X) = Number of characters in the buffer.

Examples

1. Data: Line (from keyboard is 'ENTERcr'

   Result: Buffer index = 5 (length of line)
   Buffer contains 'ENTER'
   'ENTER' echoed to terminal, followed by the new line sequence (typically either carriage return, line feed or just carriage return)
   Note that the 'cr' (carriage return) character does not appear in the buffer.

2. Data: Line (from keyboard) is 'DMcontrolHN controlXENTETcontrolHRcr'.

   Result: Buffer index = 5 (length of actual line)
   Buffer contains 'ENTER'
   'ENTER' echoed to terminal, followed by the new line sequence (typically either carriage return, line feed or just carriage return)

The sequence of operations is as follows:

Character Typed	Initial Buffer	Final Buffer
D	empty	'D'
M	'D'	'DM'
control H	'DM'	'D'
N	'D'	'DN'
control X	'DN'	empty
E	empty	'E'
N	'E'	'EN'
T	'EN'	'ENT'
E	'ENT'	'ENTE'
T	'ENTE'	'ENTET'
control H	'ENTET'	'ENTE'
R	'ENTE'	'ENTER'
cr	'ENTER'	'ENTER'

What has happened is the following:

a. The operator types 'D', 'M'

b. The operator recognizes that 'M' is incorrect (should be 'N'), types control H to delete it, and types 'N'.

c. The operator then recognizes that the initial 'D' is incorrect also (should be 'E'). Since the character to be deleted is not the latest one, the operator types control X to delete the entire line, and then types 'ENTET'.

d. The operator recognizes that the second 'T' is incorrect (should be 'R'), types control H to delete it, and types 'R'.

e. The operator types a carriage return to conclude the line.

```
;       Title              Read line                                    ;
;       Name:              RDLINE                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Purpose:           Read characters from the input device until  ;
;                          a carriage return is found. RDLINE defines the ;
;                          following control characters:                ;
;                            Control H -- Delete the previous character. ;
;                            Control X -- Delete all characters.        ;
;                                                                       ;
;       Entry:             Register A = High byte of buffer address     ;
;                          Register Y = Low byte of buffer address      ;
;                          Register X = Length of the buffer            ;
;                                                                       ;
;       Exit:              Register X = Number of characters in the buffer ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:              Not applicable.                              ;
;                                                                       ;
;       Size:              Program 138 bytes                            ;
;                          Data      4 bytes plus                       ;
;                                    2 bytes in page zero               ;
;                                                                       ;
;                                                                       ;
;

;PAGE ZERO POINTER
BUFADR  .EQU    0D0H    ;INPUT BUFFER ADDRESS

;EQUATES
DELKEY  .EQU    018H    ;DELETE LINE KEYBOARD CHARACTER
BSKEY   .EQU    08H     ;BACKSPACE KEYBOARD CHARACTER
```

```
CRKEY   .EQU    0DH         ;CARRIAGE RETURN KEYBOARD CHARACTER
SPACE   .EQU    020H        ;SPACE CHARACTER
BELL    .EQU    07H         ;BELL CHARACTER TO RING THE BELL ON THE TERMINAL


RDLINE:
        ;SAVE PARAMETERS
        STA     BUFADR+1            ;SAVE HIGH BYTE OF INPUT BUFFER ADDRESS
        STY     BUFADR              ;SAVE LOW BYTE OF INPUT BUFFER ADDRESS
        STX     BUFLEN              ;SAVE MAXIMUM LENGTH

        ;INITIALIZE BUFFER INDEX TO ZERO
INIT:
        LDA     #0
        STA     BUFIDX

        ;READ LOOP
        ;READ CHARACTERS UNTIL A CARRIAGE RETURN OCCURS
RDLOOP:
        JSR     RDCHAR              ;READ A CHARACTER FROM THE KEYBOARD
                                    ;DOES NOT ECHO

        ;CHECK FOR CARRIAGE RETURN AND EXIT IF FOUND
        CMP     #CRKEY
        BEQ     EXITRD

        ;CHECK FOR BACKSPACE AND BACK UP IF FOUND
        CMP     #BSKEY
        BNE     RDLP1               ;BRANCH IF NOT BACKSPACE CHARACTER
        JSR     BACKSP              ;IF BACKSPACE, BACK UP ONE CHARACTER
        JMP     RDLOOP              ; THEN START READ LOOP AGAIN

        ;CHECK FOR DELETE LINE CHARACTER AND DELETE LINE IF FOUND
RDLP1:
        CMP     #DELKEY
        BNE     RDLP2               ;BRANCH IF NOT DELETE LINE CHARACTER
DEL1:
        JSR     BACKSP              ;DELETE A CHARACTER
        LDA     BUFIDX              ;CONTINUE DELETING UNTIL BUFFER IS EMPTY
        BNE     DEL1
        BEQ     RDLOOP              ;THEN GO READ THE NEXT CHARACTER

        ;NOT A SPECIAL CHARACTER
        ; CHECK IF BUFFER IS FULL
        ; IF NOT FULL STORE CHARACTER AND ECHO
RDLP2:
        LDY     BUFIDX              ;IS BUFFER FULL?
        CPY     BUFLEN
        BCC     STRCH               ;BRANCH IF NOT
        LDA     #BELL               ;YES IT IS FULL, RING THE TERMINAL'S BELL
        JSR     WRCHAR
        JMP     RDLOOP              ;THEN CONTINUE THE READ LOOP

STRCH:
        STA     (BUFADR),Y          ;STORE THE CHARACTER
        JSR     WRCHAR              ;ECHO CHARACTER TO TERMINAL
```

```
        INC     BUFIDX          ;INCREMENT BUFFER INDEX
        JMP     RDLOOP          ;THEN CONTINUE THE READ LOOP

        ;EXIT SEQUENCE
        ;ECHO NEW LINE SEQUENCE (USUALLY CR,LF)
        ;GET LENGTH OF BUFFER
EXITRD:
        JSR     WRNEWL          ;ECHO THE NEW LINE SEQUENCE
        LDX     BUFIDX          ;RETURN THE LENGTH IN X
        RTS                     ;RETURN


;*********************************************
;
; THE FOLLOWING SUBROUTINES ARE SYSTEM SPECIFIC,
; THE APPLE II WAS USED IN THESE EXAMPLES.
;
;*********************************************

;*********************************************
;ROUTINE: RDCHAR
;PURPOSE: READ A CHARACTER BUT DO NOT ECHO TO OUTPUT DEVICE
;ENTRY: NONE
;EXIT:  REGISTER A = CHARACTER
;REGISTERS USED: ALL
;****************************************

RDCHAR:
        JSR     0FD0CH          ;APPLE MONITOR READ KEYBOARD
        AND     #01111111B      ;ZERO BIT 7
        RTS

;****************************************
;ROUTINE: WRCHAR
;PURPOSE: WRITE A CHARACTER TO THE OUTPUT DEVICE
;ENTRY: REGISTER A = CHARACTER
;EXIT:  NONE
;REGISTERS USED: ALL
;****************************************

WRCHAR:
        ORA     #10000000B      ;SET BIT 7
        JSR     0FDEDH          ;APPLE MONITOR CHARACTER OUTPUT ROUTINE
        RTS

;****************************************
;ROUTINE: WRNEWL
;PURPOSE: ISSUE THE APPROPRIATE NEW LINE CHARACTER OR
;         CHARACTERS. NORMALLY, THIS IS A CARRIAGE RETURN
;         AND LINE FEED, BUT SOME COMPUTERS (SUCH AS APPLE II)
;         REQUIRE ONLY A CARRIAGE RETURN.
;ENTRY: NONE
;EXIT:  NONE
;REGISTERS USED: ALL
;*******************************************
```

```
WRNEWL:
        JSR     0FD8EH          ;ECHO CARRIAGE RETURN AND LINE FEED
        RTS

;***************************************
;ROUTINE: BACKSP
;PURPOSE: PERFORM A DESTRUCTIVE BACKSPACE
;ENTRY: BUFIDX = INDEX TO NEXT AVAILABLE LOCATION IN BUFFER
;EXIT:  CHARACTER REMOVED FROM BUFFER
;REGISTERS USED: ALL
;***************************************

BACKSP:

        ;CHECK FOR EMPTY BUFFER
        LDA     BUFIDX
        BEQ     EXITBS          ;EXIT IF NO CHARACTERS IN BUFFER

        ;BUFFER IS NOT EMPTY SO DECREMENT BUFFER INDEX
        DEC     BUFIDX          ;DECREMENT BUFFER INDEX

        ;OUTPUT BACKSPACE STRING
        LDA     #LENBSS
        STA     COUNT           ;COUNT = LENGTH OF BACKSPACE STRING
        LDA     #0
        STA     INDEX           ;INDEX = INDEX TO FIRST CHARACTER
BSLOOP:
        LDA     COUNT
        BEQ     EXITBS          ;EXIT IF ALL CHARACTERS HAVE BEEN SENT
        LDY     INDEX
        LDA     BSSTRG,Y        ;GET NEXT CHARACTER
        JSR     WRCHAR          ;OUTPUT CHARACTER
        INC     INDEX
        DEC     COUNT
        JMP     BSLOOP

EXITBS:
        RTS

CSRLFT  .EQU    08H     ;CHARACTER WHICH MOVES CURSOR LEFT ONE LOCATION
LENBSS: .EQU    3       ;LENGTH OF BACKSPACE STRING
BSSTRG: .BYTE   CSRLFT,SPACE,CSRLFT


;DATA
BUFIDX: .BLOCK  1               ;INDEX TO NEXT AVAILABLE CHARACTER IN BUFFER
BUFLEN: .BLOCK  1               ;BUFFER LENGTH
COUNT:  .BLOCK  1               ;COUNT FOR BACKSPACE AND RETYPE
INDEX:  .BLOCK  1               ;INDEX FOR BACKSPACE AND RETYPE


;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;
```

```
SC1001:
        ;READ LINE
        LDA     #"?"
        JSR     WRCHAR          ;OUTPUT PROMPT (QUESTION MARK)
        LDA     ADRBUF+1        ;GET THE BUFFER ADDRESS
        LDY     ADRBUF
        LDX     #LINBUF         ;GET THE BUFFER LENGTH
        JSR     RDLINE          ;READ A LINE

        ;ECHO LINE
        STX     CNT             ;STORE NUMBER OF CHARACTERS IN THE BUFFER
        LDA     #0
        STA     IDX
TLOOP:
        LDA     CNT
        BNE     TLOOP1          ;BRANCH IF THERE ARE MORE CHARACTERS TO SEND
        JSR     WRNEWL          ;IF NOT ISSUE NEW LINE (CR,LF)
        JMP     SC1001          ;AND START OVER

TLOOP1:
        LDY     IDX
        LDA     INBUFF,Y        ;GET THE NEXT CHARACTER
        JSR     WRCHAR          ;OUTPUT IT
        INC     IDX
        DEC     CNT             ;DECREMENT LOOP COUNTER
        JMP     TLOOP


;DATA SECTION
IDX:     .BLOCK  1              ;INDEX
CNT:     .BLOCK  1              ;COUNTER
ADRBUF:  .WORD   INBUFF         ;ADDRESS OF INPUT BUFFER
LINBUF:  .EQU    10H            ;LENGTH OF INPUT BUFFER
INBUFF:  .BLOCK  LINBUF         ;DEFINE THE INPUT BUFFER


         .END    ;PROGRAM
```

Write a Line of Characters to an Output Device (WRLINE)

10B

Registers Used: All

Execution Time: 24 cycles overhead plus 25 cycles per byte (besides the execution time of subroutine WRCHAR).

Program Size: 37 bytes

Data Memory Required: Two bytes anywhere in RAM plus two bytes on page 0. The two bytes anywhere in RAM hold the buffer index (one byte at address BUFIDX) and the buffer length (one byte at address BUFLEN). The two bytes on page 0 hold a pointer to the output buffer (starting at address BUFADR, $00D0_{16}$ in the listing).

Special Case:

A buffer length of zero causes an immediate exit with no characters sent to the output device.

Writes characters to an output device using the computer-dependent subroutine WRCHAR, which writes the character in the accumulator on the output device. Continues until it empties a buffer with given length and starting address. This subroutine is intended as an example of a typical output driver. The specific I/O subroutines will, of course, be computer-dependent. The specific example described is the Apple II computer with the following features:

1. The entry point for the routine that sends a character to the monitor is $FDED_{16}$.

2. The character to be written must be placed in the accumulator with bit 7 set to 1.

Procedure: The program exits immediately if the buffer length is zero. Otherwise, the program sends characters to the output device one at a time until the buffer is emptied. The program saves all its temporary data in memory rather than in registers to avoid dependence on the WRCHAR routine.

Entry Conditions

(A) = More significant byte of starting address of buffer

(Y) = Less significant byte of starting address of buffer

(X) = Length (size) of the buffer in bytes.

Exit Conditions

None

Example

Data: Buffer length = 5
Buffer contains 'ENTER'

Result: 'ENTER' sent to the output device.

```
;       Title           Write line                                          ;
;       Name:           WRLINE                                              ;
;                                                                           ;
;                                                                           ;

;                                                                           ;
;       Purpose:        Write characters to the output device               ;
;                                                                           ;
;       Entry:          Register A = High byte of buffer address            ;
;                       Register Y = Low byte of buffer address             ;
;                       Register X = Length of the buffer in bytes          ;
;                                                                           ;
;       Exit:           None                                                ;
;                                                                           ;
;       Registers used: All                                                 ;
;                                                                           ;
;       Time:           24 cycles overhead plus                             ;
;                       (25 + execution time of WRCHAR) cycles per byte    ;
;                                                                           ;
;       Size:           Program 37 bytes                                    ;
;                       Data     2 bytes plus                               ;
;                                2 bytes in page zero                       ;
;                                                                           ;
;                                                                           ;
;                                                                           ;


;PAGE ZERO POINTER
BUFADR  .EQU    0D0H                ;OUTPUT BUFFER ADDRESS

WRLINE:
        ;SAVE PARAMETERS
        STA     BUFADR+1            ;SAVE HIGH BYTE OF OUTPUT BUFFER ADDRESS
        STY     BUFADR              ;SAVE LOW BYTE OF OUTPUT BUFFER ADDRESS
        STX     BUFLEN              ;SAVE LENGTH
        BEQ     EXIT                ;EXIT IF LENGTH = 0

        ;INITIALIZE BUFFER INDEX TO ZERO
        LDA     #0
        STA     BUFIDX
WRLOOP:
        LDY     BUFIDX
        LDA     (BUFADR),Y          ;GET NEXT CHARACTER
        JSR     WRCHAR              ;OUTPUT CHARACTER
        INC     BUFIDX              ;INCREMENT BUFFER INDEX
        DEC     BUFLEN              ;DECREMENT BUFFER LENGTH
        BNE     WRLOOP              ;BRANCH IF NOT DONE

EXIT:
        RTS
;**********************************************
;
; THE FOLLOWING SUBROUTINES ARE SYSTEM SPECIFIC,
; THE APPLE II WAS USED IN THIS EXAMPLE.
;
;**********************************************
```

```
;****************************************
;ROUTINE: WRCHAR
;PURPOSE: WRITE A CHARACTER TO THE OUTPUT DEVICE
;ENTRY: REGISTER A = CHARACTER
;EXIT:  NONE
;REGISTERS USED: ALL
;****************************************

WRCHAR:
        ORA     #10000000B      ;SET BIT 7
        JSR     0FDEDH          ;APPLE MONITOR CHARACTER OUTPUT ROUTINE
        RTS

;********************************
;
; DATA SECTION
BUFIDX: .BLOCK  1               ;INDEX TO NEXT AVAILABLE CHARACTER IN BUFFER
BUFLEN: .BLOCK  1               ;BUFFER LENGTH


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC1002:
        ;READ LINE USING THE APPLE MONITOR GETLN ROUTINE AT 0FD6AH
        ; 33H = ADDRESS CONTAINING APPLE PROMPT CHARACTER
        ; 200H = BUFFER ADDRESS
        LDA     #"?" OR 80H     ;USE ? FOR PROMPT WITH BIT 7 SET
        STA     033H            ;SET UP APPLE PROMPT CHARACTER
        JSR     0FD6AH          ;CALL APPLE MONITOR GETLN ROUTINE
        STX     LENGTH          ;RETURN LENGTH IN REGISTER X

        ;WRITE THE LINE
        LDA     #02H            ;A = HIGH BYTE OF BUFFER ADDRESS
        LDY     #0              ;Y = LOW BYTE OF BUFFER ADDRESS
        LDX     LENGTH          ;X = LENGTH OF BUFFER
        JSR     WRLINE          ;OUTPUT THE BUFFER
        JSR     0FD8EH          ;OUTPUT CARRIAGE RETURN VIA APPLE MONITOR

        JMP     SC1002          ;CONTINUE

;DATA SECTION
LENGTH: .BLOCK  1


        .END    ;PROGRAM
```

Generate Even Parity (GEPRTY) 10C

Generates even parity for a seven-bit character and places it in bit 7. Even parity for a seven-bit character is a bit that makes the total number of 1 bits in the byte even.

Procedure: The program generates even parity by counting the number of 1 bits in the seven least significant bits of the accumulator. The counting is accomplished by shifting the data left logically and incrementing the count by one if the bit shifted into the Carry is 1. The least significant bit of the count is an even parity bit; the program concludes by shifting that bit to the Carry and then to bit 7 of the original data.

Registers Used: A, F

Execution Time: 114 cycles maximum. Depends on the number of 1 bits in the data and how rapidly the series of logical shifts makes the data zero. The program exits as soon as the remaining bits of data are all zeros, so the execution time is shorter if the less significant bits are all zeros.

Program Size: 39 bytes

Data Memory Required: One byte anywhere in RAM (at address VALUE) for the data.

Entry Conditions

Data in the accumulator (bit 7 is not used).

Exit Conditions

Data with even parity in bit 7 in the accumulator.

Examples

1. Data: (A) = 42_{16} = 01000010_2 (ASCII B)

   Result: (A) = 42_{16} = 01000010_2 (ASCII B with bit 7 cleared)
   Even parity is 0, since 01000010_2 has an even number (2) of 1 bits.

2. Data: (A) = 43_{16} = 01000011_2 (ASCII C)

   Result: (A) = $C3_{16}$ = 11000011_2 (ASCII C with bit 7 set)

```
;       Title           Generate even parity                          ;
;       Name:           GEPRTY                                        ;
;                                                                     ;
;                                                                     ;
;                                                                     ;
;       Purpose:        Generate even parity in bit 7 for a 7-bit     ;
;                       character.                                    ;
;                                                                     ;
;       Entry:          Register A = Character                        ;
```

```
;       Exit:               Register A = Character with even parity    ;
;                                                                      ;
;       Registers used: A,F                                            ;
;                                                                      ;
;       Time:               114 cycles maximum                         ;
;                                                                      ;
;       Size:               Program 39 bytes                           ;
;                           Data     1 byte                            ;
;                                                                      ;
;                                                                      ;

GEPRTY:
        ;SAVE THE DATA
        STA     VALUE

        ;SAVE X AND Y REGISTERS
        PHA
        TXA
        PHA
        TYA

        ;COUNT THE NUMBER OF 1 BITS IN BITS 0 THROUGH 6 OF THE DATA
        LDY     #0      ;INITIALIZE NUMBER OF 1 BITS TO ZERO
        LDA     VALUE   ;GET DATA
        ASL     A       ;DROP BIT 7 OF THE DATA, NEXT BIT TO BIT 7
        STA     VALUE
GELOOP: BPL     SHFT    ;BRANCH IF NEXT BIT (BIT 7) IS 0
        INY             ;ELSE INCREMENT NUMBER OF 1 BITS
SHFT:   ASL     A
        BNE     GELOOP  ;BRANCH IF THERE ARE MORE 1 BITS IN THE BYTE

        TYA             ;BIT 0 OF NUMBER OF 1 BITS IS EVEN PARITY
        LSR     A       ;MOVE PARITY TO CARRY
        LDA     VALUE
        ROR     A       ;ROTATE ONCE TO FORM BYTE WITH PARITY IN BIT 7
        STA     VALUE

        ;RESTORE X AND Y AND EXIT
        PLA
        TAY
        PLA
        TAX
        LDA     VALUE   ;GET VALUE WITH PARITY
        RTS             ;RETURN

;DATA SECTION
VALUE:  .BLOCK  1       ;TEMPORARY DATA STORAGE

;                                                                      ;
;                                                                      ;
;       SAMPLE EXECUTION:                                              ;
;                                                                      ;
;                                                                      ;
```

```
;GENERATE PARITY FOR VALUES FROM 0..127 AND STORE THEM IN BUFFER
SC1003:
        LDX     #0
SC1LP:
        TXA
        JSR     GEPRTY          ;GENERATE EVEN PARITY
        STA     BUFFER,X        ;STORE THE VALUE WITH EVEN PARITY
        INX
        CPX     #80H
        BNE     SC1LP           ;BRANCH IF NOT DONE
        BRK

BUFFER  .BLOCK  128


        .END    ;PROGRAM
```

Sets the Carry flag to 0 if a data byte has even parity and to 1 if it has odd parity. A byte has even parity if it has an even number of 1 bits and odd parity if it has an odd number of 1 bits.

Procedure: The program counts the number of 1 bits in the data by shifting the data left logically and incrementing a count if the bit shifted into the Carry is 1. The program quits as soon as the shifted data becomes zero (since zero obviously does not contain any 1 bits). The least significant bit of the count is 0 if the data byte contains an even number of 1 bits and 1 if the data byte contains an odd number of 1 bits. The program concludes by shifting the least significant bit of the count to the Carry flag.

Registers Used: A, F

Execution Time: 111 cycles maximum. Depends on the number of 1 bits in the data and how rapidly the series of logical shifts makes the data zero. The program exits as soon as the remaining bits of data are all zeros, so the execution time is shorter if the less significant bits are all zeros.

Program Size: 25 bytes

Data Memory Required: One byte anywhere in RAM (at address VALUE) for the data.

Entry Conditions

Data byte in the accumulator (bit 7 is included in the parity generation).

Exit Conditions

Carry = 0 if the parity of the data byte is even, 1 if the parity is odd.

Examples

1. Data: (A) = 42_{16} = 01000010_2 (ASCII B)

   Result: Carry = 0, since 42_{16} (01000010_2) has an even number (2) of 1 bits.

2. Data: A) = 43_{16} = 01000011_2 (ASCII C)

   Result: Carry = 1, since 43_{16} (01000011_2) has an odd number (3) of 1 bits.

```
;       Title              Check parity                          ;
;       Name:              CKPRTY                                ;
;                                                                ;
;                                                                ;

;                                                                ;
;       Purpose:           Check parity of a byte                ;
;                                                                ;
;       Entry:             Register A = Byte with parity in bit 7;
;                                                                ;
;       Exit:              Carry = 0 if parity is even.          ;
;                          Carry = 1 if parity is odd.           ;
;                                                                ;
;       Registers used: A,F                                      ;
;                                                                ;
;       Time:              111 cycles maximum                    ;
;                                                                ;
;       Size:              Program 25 bytes                      ;
;                          Data     1 byte                       ;
;                                                                ;
;                                                                ;

CKPRTY:
        ;SAVE DATA VALUE
        STA     VALUE

        ;SAVE REGISTERS X AND Y
        TXA
        PHA
        TYA
        PHA

        ;COUNT THE NUMBER OF 1 BITS IN THE VALUE
        LDY     #0      ;NUMBER OF 1 BITS = 0
        LDA     VALUE
CKLOOP: BPL     SHFT    ;BRANCH IF NEXT BIT = 0 (BIT 7)
        INY             ;ELSE INCREMENT NUMBER OF 1 BITS
SHFT:   ASL     A       ;SHIFT NEXT BIT TO BIT 7
        BNE     CKLOOP  ;CONTINUE UNTIL ALL BITS ARE 0

        TYA
        LSR     A       ;CARRY FLAG = LSB OF NUMBER OF 1 BITS

        ;RESTORE REGISTERS X AND Y AND EXIT
        PLA
        TAY
        PLA
        TAX
        RTS
```

```
VALUE   .BLOCK  1       ;DATA BYTE

;                                                                     ;
;                                                                     ;
;       SAMPLE EXECUTION:                                             ;
;                                                                     ;
;                                                                     ;


;CHECK PARITY FOR VALUES FROM 0..255 AND STORE THEM IN BUFFER
;BUFFER[VALUE] = 0 FOR EVEN PARITY
;BUFFER[VALUE] = 1 FOR ODD PARITY
SC1004:
        LDX     #0
SCLP:
        TXA
        JSR     CKPRTY          ;CHECK PARITY
        LDA     #0
        ROL     A               ;GET PARITY TO BIT 0
        STA     BUFFER,X        ;STORE THE PARITY
        INX                     ;INCREMENT VALUE
        BNE     SCLP            ;CONTINUE THROUGH ALL THE VALUES
        BRK
        JMP     SC1004

BUFFER  .BLOCK  256

        .END    ;PROGRAM
```

CRC-16 Checking and Generation (ICRC16,CRC16) 10E

Generates a 16-bit cyclic redundancy check (CRC) based on the IBM Binary Synchronous Communications (BSC or Bisync) protocol. Uses the polynomial $X^{16} + X^{15} + X^2 + 1$ to generate the CRC. The entry point ICRC16 initializes the CRC to 0 and the polynomial to the appropriate bit pattern. The entry point CRC16 combines the previous CRC with the CRC generated from the next byte of data. The entry point GCRC16 returns the CRC.

Procedure: Subroutine ICRC16 initializes the CRC to zero and the polynomial to the appropriate value (one in each bit position corresponding to a power of X present in the polynomial). Subroutine CRC16 updates the CRC according to a specific byte of data. It updates the CRC by shifting the data and the CRC left one bit and exclusive-ORing the CRC with the polynomial whenever the exclusive-OR of the data bit and the most significant bit of the CRC is 1. Subroutine CRC16 leaves the CRC in memory locations CRC (less significant byte) and CRC+1 (more significant byte). Subroutine GCRC16 loads the CRC into the accumulator (more significant byte) and index register Y (less significant byte).

Registers Used:
1. By ICRC16: A, F
2. By CRC16: None
3. By GCRC16: A, F, Y

Execution Time:
1. For ICRC16: 28 cycles
2. For CRC16: 302 cycles minimum if no 1 bits are generated and the polynomial and the CRC never have to be EXCLUSIVE-ORed. 19 extra cycles for each time the polynomial and the CRC must be EXCLUSIVE-ORed. Thus, the maximum execution time is 302 + 19*8 = 454 cycles.
3. For GCRC16: 14 cycles

Program Size:
1. For ICRC16: 19 bytes
2. For CRC16: 53 bytes
3. For GCRC16: 7 bytes

Data Memory Required: Five bytes anywhere in RAM for the CRC (two bytes starting at address CRC), the polynomial (two bytes starting at address PLY), and the data byte (one byte at address VALUE).

Entry Conditions

1. For ICRC16: none

2. For CRC16: data byte in the accumulator, previous CRC in memory locations CRC (less significant byte) and CRC+1 (more significant byte), CRC polynomial in memory locations PLY (less significant byte) and PLY+1 (more significant byte)

3. For GCRC16: CRC in memory locations CRC (less significant byte), and CRC+1 (more significant byte).

Exit Conditions

1. For ICRC16: zero (initial CRC value) in memory locations CRC (less significant byte) and CRC+1 (more significant byte)

CRC polynomial in memory locations PLY (less significant byte) and PLY+1 (more significant byte)

2. For CRC16: CRC with current data byte included in memory locations CRC (less significant byte) and CRC+1 (more significant byte)

3. For GCRC16: CRC in the accumulator (more significant byte) and index register Y (less significant byte).

Examples

1. Generating a CRC.

Call ICRC16 to initialize the polynomial and start the CRC at zero.

Call CRC16 to update the CRC for each byte of data for which the CRC is to be generated.

Call GCRC16 to obtain the resulting CRC (more significant byte in A, less significant byte in Y).

2. Checking a CRC.

Call ICRC16 to initialize the polynomial and start the CRC at zero.

Call CRC16 to update the CRC for each byte of data (including the stored CRC) for checking.

Call GCRC16 to obtain the resulting CRC (more significant byte in A, less significant byte in Y). If there were no errors, both bytes should be zero.

Note that only subroutine ICRC16 depends on the particular CRC polynomial being used. To change the polynomial requires only a change of the data that ICRC16 loads into memory locations PLY (less significant byte) and PLY+1 (more significant byte).

Reference

J.E. McNamara, *Technical Aspects of Data Communications,* Digital Equipment Corp., Maynard, Mass., 1977. This book contains explanations of CRC and the various communications protocols.

```
;       Title           Generate CRC-16                                  ;
;       Name:           CRC16                                            ;
;                                                                        ;
;                                                                        ;

;                                                                        ;
;       Purpose:        Generate a 16 bit CRC based on the IBM binary    ;
;                       synchronous communications protocol. The CRC is  ;
;                       based on the following polynomial:               ;
;                         (^ indicates "to the power")                   ;
;                              X^16 + X^15 + X^2 +1                      ;
;                                                                        ;
;                       To generate a CRC:                               ;
;                           1)  Call ICRC16 to initialize the CRC to 0   ;
;                               and the CRC polynomial.                  ;
;                           2)  Call CRC16 for each byte of data for     ;
;                               which the CRC is to be generated.        ;
;                           3)  Call GCRC16 to get the resulting CRC.    ;
;                               It should then be appended to the data,  ;
;                               high byte first.                         ;
;                                                                        ;
;                       To check a CRC:                                  ;
;                           1)  Call ICRC16 to initialize the CRC.       ;
;                           2)  Call CRC16 for each byte of data and     ;
;                               the 2 bytes of CRC previously generated.;
;                           3)  Call GCRC16 to obtain the CRC. It will   ;
;                               be zero if no errors have occurred.      ;
;                                                                        ;
;       Entry:          Register A = Data byte                           ;
;                                                                        ;
;       Exit:           CRCLO and CRCHI updated                          ;
;                       Register A = Data byte                           ;
;                                                                        ;
;       Registers used: None                                             ;
;                                                                        ;
;       Time:           302 cycles minimum if no 1 bits are generated.   ;
;                       454 cycles maximum if all 1 bits are generated.  ;
;                                                                        ;
;       Size:           Program 53 bytes                                 ;
;                       Data     5 bytes                                 ;
;                                                                        ;
;                                                                        ;

CRC16:
        ;SAVE THE DATA BYTE
        STA     VALUE

        ;SAVE ALL REGISTERS
        PHP
        PHA
        TYA
        PHA
        TXA
        PHA
```

```
        ;LOOP THROUGH EACH BIT GENERATING THE CRC
        LDX     #8                  ;8 BITS PER BYTE
CRCLP:
        ASL     VALUE               ;MOVE BIT 7 TO CARRY
        ROR     A                   ;MOVE CARRY TO BIT 7
        AND     #10000000B          ;MASK OFF ALL OTHER BITS
        EOR     CRC+1               ;EXCLUSIVE OR BIT 7 WITH BIT 16 OF THE CRC
        ASL     CRC                 ;SHIFT CRC LEFT 1 BIT (FIRST THE LOW BYTE,
        ROL     A                   ; THEN THE HIGH BYTE)
        BCC     CRCLP1              ;BRANCH IF THE MSB OF THE CRC IS 1

        ;BIT 7 IS 1 SO EXCLUSIVE-OR THE CRC WITH THE POLYNOMIAL
        TAY                         ;SAVE CRC HIGH IN Y
        LDA     CRC
        EOR     PLY                 ;EXCLUSIVE OR LOW BYTE WITH THE POLYNOMIAL
        STA     CRC
        TYA
        EOR     PLY+1               ;DO HIGH BYTE ALSO

CRCLP1:
        STA     CRC+1               ;STORE THE HIGH BYTE OF THE CRC
        DEX
        BNE     CRCLP               ;BRANCH IF NOT DONE WITH ALL 8 BITS

        ;RESTORE THE REGISTERS AND EXIT
        PLA
        TAX
        PLA
        TAY
        PLA
        PLP
        RTS

;*******************************************
;ROUTINE: ICRC16
;PURPOSE: INITIALIZE CRCHI, CRCLO, PLYHI, PLYLO
;ENTRY: NONE
;EXIT:  CRC AND POLYNOMIAL INITIALIZED
;REGISTERS USED: A,F
;*******************************************

ICRC16:
        LDA     #0
        STA     CRC                 ;CRC = 0
        STA     CRC+1
        LDA     #5
        STA     PLY                 ;PLY = 8005H
                                    ;8005H IS FOR X^16+X^15+X^2+1
                                    ; (1 IN EACH POSITION FOR WHICH A POWER
                                    ;  APPEARS IN THE FORMULA)
        LDA     #80H
        STA     PLY+1
        RTS
```

```
;*******************************************
;ROUTINE: GCRC16
;PURPOSE: GET THE CRC16 VALUE
;ENTRY: NONE
;EXIT:  REGISTER A = CRC16 HIGH BYTE
;       REGISTER Y = CRC16 LOW BYTE
;REGISTERS USED: A,F,Y
;*******************************************

GCRC16:
        LDA     CRC+1           ;A = HIGH BYTE
        LDY     CRC             ;Y = LOW BYTE
        RTS

VALUE:  .BLOCK  1               ;DATA BYTE
CRC:    .BLOCK  2               ;CRC VALUE
PLY:    .BLOCK  2               ;POLYNOMIAL VALUE USED TO GENERATE THE CRC


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


        ;GENERATE A CRC FOR A VALUE OF 1 AND CHECK IT
SC1005:
        JSR     ICRC16
        LDA     #1
        JSR     CRC16           ;GENERATE CRC
        JSR     GCRC16
        TAX                     ;SAVE CRC HIGH BYTE IN REGISTER X
        JSR     ICRC16          ;INITIALIZE AGAIN
        LDA     #1
        JSR     CRC16           ;CHECK CRC BY GENERATING IT FOR DATA
        TXA
        JSR     CRC16           ; AND THE STORED CRC ALSO
        TYA
        JSR     CRC16
        JSR     GCRC16
        BRK                     ;THE CRC SHOULD BE ZERO IN REGISTERS A AND Y

        ;GENERATE A CRC FOR THE VALUES FROM 0..255 AND CHECK IT
        JSR     ICRC16
        LDX     #0
GENLP:
        TXA                     ;GET NEXT BYTE
        JSR     CRC16           ;UPDATE CRC
        INX
        BNE     GENLP           ;BRANCH IF NOT DONE

        JSR     GCRC16          ;GET RESULTING CRC
        STA     CRCVAL+1        ;AND SAVE IT
        STY     CRCVAL
```

```
        ;CHECK THE CRC BY GENERATING IT AGAIN
        JSR     ICRC16
        LDX     #0
CHKLP:
        TXA
        JSR     CRC16
        INX
        BNE     CHKLP

        ;ALSO INCLUDE STORED CRC IN CHECK
        LDA     CRCVAL+1
        JSR     CRC16           ;HIGH BYTE OF CRC FIRST
        LDA     CRCVAL
        JSR     CRC16           ;THEN LOW BYTE OF CRC

        JSR     GCRC16          ;GET RESULTING CRC
        BRK                     ;IT SHOULD BE 0
        JMP     SC1005

CRCVAL: BLOCK   2

        .END
```

I/O Device Table Handler (IOHDLR)

Performs input and output in a device-independent manner using I/O control blocks and an I/O device table. The I/O device table consists of a linked list; each entry contains a link to the next entry, the device number, and starting addresses for routines that initialize the device, determine its input status, read data from it, determine its output status, and write data to it. An I/O control block is an array containing the device number, the operation number, device status, the starting address of the device's buffer, and the length of the device's buffer. The user must provide IOHDLR with the address of an appropriate I/O control block and the data if only one byte is to be written. IOHDLR will return a copy of the status byte and the data if only one byte is read.

This subroutine is intended as an example of how to handle input and output in a device-independent manner. The I/O device table must be constructed using subroutines INITIO, which initializes the device list to empty, and ADDDL, which adds a device to the list. A specific example for the Apple II sets up the Apple II console as device 1 and the printer as device 2; a test routine reads a line from the console and echoes it to the console and the printer.

A general purpose program will perform input or output by obtaining or constructing an I/O control block and then calling IOHDLR. Subroutine IOHDLR will then determine which device to use and how to transfer control to its I/O driver by using the I/O device table.

Procedure: The program first initializes the status byte to zero, indicating no errors. It then searches the device table, looking for the device number in the I/O control block. If it does not find a match in the table, it exits with an appropriate error number in the status byte. If the program finds a device with the proper device number, it checks for a valid operation and transfers control to the appropriate routine from the entry in the device table. That routine must then transfer control back to the original calling routine. If the operation is invalid (the operation number is too large or the starting address for the routine is zero), the program returns with an error indication in the status byte.

Registers Used

1. By IOHDRL: All
2. By INITL: A, F
3. By ADDDL: All

Execution Time

1. For IOHDLR: 93 cycles overhead plus 59 cycles for each unsuccessful match of a device number
2. For INITL: 14 cycles
3. For ADDDL: 48 cycles

Program Size

1. For IOHDLR: 101 bytes
2. For INITL: 9 bytes
3. For ADDDL: 21 bytes

Data Memory Required: Three bytes anywhere in RAM plus six bytes on page 0. The three bytes anywhere in RAM hold an indirect address used to vector to an I/O subroutine (two bytes starting at address OPADR) and the X register (one byte at address SVXREG). The six bytes on page 0 hold the starting address of the I/O control block (two bytes starting at address IOCB), the head of the list of devices (two bytes starting at address DVLST), and the starting address of the current device table entry (two bytes starting at address CURDEV).

Subroutine INITDL initializes the device list, setting the initial link to zero.

Subroutine ADDDL adds an entry to the device list, making its address the head of the list and setting its link field to the old head of the list.

Entry Conditions

1. For IOHDLR:

(A) = More significant byte of starting address of input/output control block

(Y) = Less significant byte of starting address of input/output control block

(X) = Byte of data if the operation is to write one byte.

2. For INITL: None

3. For ADDDL:

(A) = More significant byte of starting address of a device table entry

(Y) = Less significant byte of starting address of a device table entry.

Exit Conditions

1. For IOHDLR:

(A) = I/O control block status byte if an error is found; otherwise, the routine exits to the appropriate I/O driver.

(X) = Byte of data if the operation is to read one byte.

2. For INITL:

Device list header (addresses DVLST and DVLST+1) cleared to indicate empty list.

3. For ADDDL:

Device table entry added to list.

Example

In the example provided, we have the following structure:

INPUT/OUTPUT OPERATIONS

Operation Number	Operation
0	Initialize device
1	Determine input status
2	Read 1 byte from input device
3	Read N bytes from input device (normally one line)
4	Determine output status
5	Write one byte to output device
6	Write N bytes to output device (normally one line)

INPUT/OUTPUT CONTROL BLOCK

Index	Contents
0	Device number
1	Operation number
2	Status
3	Less significant byte of starting address of buffer
4	More significant byte of starting address of buffer
5	Less significant byte of buffer length
6	More significant byte of buffer length

DEVICE TABLE ENTRY

Index	Contents
0	Less significant byte of link field (starting address of next element)
1	More significant byte of link field (starting address of next element)
2	Device number
3	Less significant byte of starting address of device initialization routine
4	More significant byte of starting address of device initialization routine
5	Less significant byte of starting address of input status determination routine
6	More significant byte of starting address of input status determination routine
7	Less significant byte of starting address of input driver routine (read 1 byte only)
8	More significant byte of starting address of input driver routine (read 1 byte only)
9	Less significant byte of starting address of input driver routine (N bytes or 1 line)
10	More significant byte of starting address of input driver routine (N bytes or 1 line)
11	Less significant byte of starting address of output status determination routine
12	More significant byte of starting address of output status determination routine
13	Less significant byte of starting address of output driver routine (write 1 byte only)
14	More significant byte of starting address of output driver routine (write 1 byte only)
15	Less significant byte of starting address of output driver routine (N bytes or 1 line)
16	More significant byte of starting address of output driver routine (N bytes or 1 line)

If an operation is irrelevant or undefined for a particular device (e.g., output status determination for a keyboard or an input driver routine for a printer), the corresponding starting address in the device table must be set to zero (i.e., 0000_{16}).

STATUS VALUES

Value	Description
0	No errors
1	Bad device number (no such device)
2	Data available from input device, no such operation for I/O
3	Output device ready

```
;        Title            I/O Device table handler                     ;
;        Name:            IOHDLR                                       ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;        Purpose:         Perform I/O in a device independent manner.  ;
;                         This can only be implemented by accessing all ;
;                         devices  in the same way using a I/O Control ;
;                         Block (IOCB) and a device table. The routines ;
;                         here will allow the following operations:    ;
;                                                                      ;
```

```
;                         Operation number  Description                  ;
;                             0            Initialize device             ;
;                             1            Input status                  ;
;                             2            Read 1 byte                   ;
;                             3            Read N bytes                  ;
;                             4            Output status                 ;
;                             5            Write 1 byte                  ;
;                             6            Write N bytes                 ;
;                                                                        ;
;                         Other operations that could be included are    ;
;                         Open, Close, Delete, Rename, and Append which  ;
;                         would support devices such as floppy disks.    ;
;                                                                        ;
;                         A IOCB will be an array of the following form: ;
;                                                                        ;
;                         IOCB + 0 = Device number                       ;
;                         IOCB + 1 = Operation number                    ;
;                         IOCB + 2 = Status                              ;
;                         IOCB + 3 = Low byte buffer address             ;
;                         IOCB + 4 = High byte of buffer address         ;
;                         IOCB + 5 = Low byte of buffer length           ;
;                         IOCB + 6 = High byte of buffer length          ;
;                                                                        ;
;                         The device table is implemented as a linked    ;
;                         list. Two routines maintain the list: INITIO,  ;
;                         which initializes the device list to empty, and ;
;                         ADDDL, which adds a device to the list.        ;
;                         A device table entry has the following form:   ;
;                                                                        ;
;                         DVTBL + 0 = Low byte of link field             ;
;                         DVTBL + 1 = High byte of link field            ;
;                         DVTBL + 2 = Device number                      ;
;                         DVTBL + 3 = Low byte of initialize device      ;
;                         DVTBL + 4 = High byte of initialize device     ;
;                         DVTBL + 5 = Low byte of input status routine   ;
;                         DVTBL + 6 = High byte of input status routine  ;
;                         DVTBL + 7 = Low byte of input 1 byte routine   ;
;                         DVTBL + 8 = High byte of input 1 byte routine  ;
;                         DVTBL + 9 = Low byte of input N bytes routine  ;
;                         DVTBL + 10= High byte of input N bytes routine ;
;                         DVTBL + 11= Low byte of output status routine  ;
;                         DVTBL + 12= High byte of output status routine ;
;                         DVTBL + 13= Low byte of output 1 byte routine  ;
;                         DVTBL + 14= High byte of output 1 byte routine ;
;                         DVTBL + 15= Low byte of output N bytes routine ;
;                         DVTBL + 16= High byte of output N bytes routine ;
;                                                                        ;
;       Entry:            Register A = High byte of IOCB                 ;
;                         Register Y = Low byte of IOCB                  ;
;                         Register X = For write 1 byte contains the byte ;
;                                      to write, a buffer is not used.   ;
;                                                                        ;
;       Exit:             Register A = a copy of the IOCB status byte    ;
;                         Register X = For read 1 byte contains the byte ;
;                                      read, a buffer is not used.       ;
;                         Status byte of IOCB is 0 if the operation was  ;
```

```
;                         completed successfully; otherwise it contains   ;
;                         the error number.                               ;
;                                                                         ;
;                         Status value    Description                     ;
;                             0           No errors                       ;
;                             1           Bad device number               ;
;                             2           Input data available, no such   ;
;                                         operation                       ;
;                             3           Output ready                    ;
;                                                                         ;
;                                                                         ;
;       Registers used: All                                               ;
;                                                                         ;
;       Time:             93 cycles minimum plus 59 cycles for each       ;
;                         device in the list which is not the requested   ;
;                         device.                                         ;
;                                                                         ;
;       Size:             Program 131 bytes                               ;
;                         Data      3 bytes plus                          ;
;                                   6 bytes in page zero                  ;
;                                                                         ;
;                                                                         ;


;IOCB AND DEVICE TABLE EQUATES
IOCBDN: .EQU    0          ;IOCB DEVICE NUMBER
IOCBOP: .EQU    1          ;IOCB OPERATION NUMBER
IOCBST: .EQU    2          ;IOCB STATUS
IOCBBA: .EQU    3          ;IOCB BUFFER ADDRESS
IOCBBL: .EQU    5          ;IOCB BUFFER LENGTH
DTLNK:  .EQU    0          ;DEVICE TABLE LINK FIELD
DTDN:   .EQU    2          ;DEVICE TABLE DEVICE NUMBER
DTSR:   .EQU    3          ;BEGINNING OF DEVICE TABLE SUBROUTINES

;OPERATION NUMBERS
NUMOP:  .EQU    7          ;NUMBER OF OPERATIONS
INIT:   .EQU    0          ;INITIALIZATION
ISTAT:  .EQU    1          ;INPUT STATUS
R1BYTE: .EQU    2          ;READ 1 BYTE
RNBYTE: .EQU    3          ;READ N BYTES
OSTAT:  .EQU    4          ;OUTPUT STATUS
W1BYTE: .EQU    5          ;WRITE 1 BYTE
WNBYTE: .EQU    6          ;WRITE N BYTES

;PAGE ZERO DEFINITIONS
IOCBA:  .EQU    0D0H       ;ADDRESS OF THE IOCB
DVLST:  .EQU    0D2H       ;ADDRESS OF A LIST OF DEVICES
CURDEV: .EQU    0D4H       ;STARTING ADDRESS OF THE CURRENT DEVICE TABLE ENTRY

IOHDLR:
        ;SAVE IOCB ADDRESS AND X REGISTER
        STA     IOCBA+1
        STY     IOCBA
        STX     SVXREG
```

```
        ;INITIALIZE STATUS BYTE TO ZERO (NO ERRORS)
        LDY     #IOCBST
        LDA     #0
        STA     (IOCBA),Y       ;STATUS := 0

        ;SEARCH DEVICE LIST FOR THIS DEVICE
        LDA     DVLST           ;START AT THE BEGINNING OF THE DEVICE LIST
        STA     CURDEV
        LDA     DVLST+1
        STA     CURDEV+1

        ;GET DEVICE NUMBER FROM IOCB TO REGISTER X
        LDY     #IOCBDN
        LDA     (IOCBA),Y
        TAX

SRCHLP:
        ;CHECK IF AT END OF DEVICE TABLE LIST (LINK FIELD = 0000)
        LDA     CURDEV
        ORA     CURDEV+1
        BEQ     BADDN           ;BRANCH IF NO MORE DEVICES

        ;CHECK IF THIS IS THE CORRECT DEVICE
        TXA
        LDY     #DTDN
        CMP     (CURDEV),Y      ;COMPARE THIS DEVICE NUMBER WITH THE REQUESTED
                                ; NUMBER
        BEQ     FOUND           ;BRANCH IF THE DEVICE IS FOUND

        ;ADVANCE TO THE NEXT DEVICE TABLE ENTRY THROUGH THE LINK FIELD
        ; MAKE CURRENT DEVICE = LINK
        LDY     #DTLNK
        LDA     (CURDEV),Y      ;GET LOW BYTE OF LINK FIELD
        PHA                     ; SAVE ON STACK
        INY
        LDA     (CURDEV),Y      ;GET HIGH BYTE OF LINK FIELD
        STA     CURDEV+1
        PLA                     ;RECOVER LOW BYTE OF LINK FIELD
        STA     CURDEV
        JMP     SRCHLP          ;CONTINUE SEARCHING

        ;FOUND THE DEVICE SO VECTOR TO THE APPROPRIATE ROUTINE IF ANY
FOUND:
        ;CHECK THAT THE OPERATION IS VALID
        LDY     #IOCBOP
        LDA     (IOCBA),Y       ;GET OPERATION NUMBER
        CMP     #NUMOP
        BCS     BADOP           ;BRANCH IF OPERATION NUMBER IS TOO LARGE

        ;GET OPERATION ADDRESS (ZERO INDICATES INVALID OPERATION)
        ASL     A               ;MULTIPLY OPERATION NUMBER BY 2 TO INDEX
        CLC                     ; ADDRESSES
        ADC     #DTSR           ;ADD TO OFFSET FOR DEVICE TABLE SUBROUTINES
        TAY                     ;USE AS INDEX INTO DEVICE TABLE
```

```
        LDA     (CURDEV),Y
        STA     OPADR           ;STORE LOW BYTE
        INY
        LDA     (CURDEV),Y
        STA     OPADR+1         ;STORE HIGH BYTE
        ORA     OPADR           ;CHECK FOR NON-ZERO OPERATION ADDRESS
        BEQ     BADOP           ;BRANCH IF OPERATION IS INVALID (ADDRESS = 0)

        LDX     SVXREG          ;RESTORE X REGISTER
        JMP     (OPADR)         ;GOTO ROUTINE

BADDN:
        LDA     #1              ;ERROR CODE 1 -- NO SUCH DEVICE
        BNE     EREXIT

BADOP:
        LDA     #2              ;ERROR CODE 2 -- NO SUCH OPERATION

EREXIT:
        LDY     #IOCBST
        STA     (IOCBA),Y       ;STORE ERROR STATUS
        RTS


;****************************************
;ROUTINE: INITDL
;PURPOSE: INITIALIZE THE DEVICE LIST TO EMPTY
;ENTRY: NONE
;EXIT:  THE DEVICE LIST SET TO NO ITEMS
;REGISTERS USED: A,F
;****************************************

INITDL:
        ;INITIALIZE DEVICE LIST TO 0 TO INDICATE NO DEVICES
        LDA     #0
        STA     DVLST
        STA     DVLST+1
        RTS


;****************************************
;ROUTINE: ADDDL
;PURPOSE: ADD A DEVICE TO THE DEVICE LIST
;ENTRY: REGISTER A = HIGH BYTE OF A DEVICE TABLE ENTRY
;       REGISTER Y = LOW BYTE OF A DEVICE TABLE ENTRY
;EXIT:  THE DEVICE TABLE ADDED TO THE DEVICE LIST
;REGISTERS USED: ALL
;****************************************

ADDDL:
        ;X,Y = NEW DEVICE TABLE ENTRY
        TAX

        ;PUSH CURRENT HEAD OF DEVICE LIST ON TO STACK
        LDA     DVLST+1
```

```
        PHA                             ;PUSH HIGH BYTE OF CURRENT HEAD OF DEVICE LIST
        LDA     DVLST
        PHA                             ;PUSH LOW BYTE ALSO

        ;MAKE NEW DEVICE TABLE ENTRY THE HEAD OF THE DEVICE LIST
        STY     DVLST
        STX     DVLST+1

        ;SET LINK FIELD OF THE NEW DEVICE TO THE OLD HEAD OF THE DEVICE LIST
        PLA
        LDY     #0
        STA     (DVLST),Y               ;STORE THE LOW BYTE
        PLA
        INY
        STA     (DVLST),Y               ;STORE THE HIGH BYTE

        RTS

;
;DATA SECTION
OPADR:  .BLOCK  2                       ;OPERATION ADDRESS USED TO VECTOR TO
                                        ; SUBROUTINE
SVXREG: .BLOCK  1                       ;TEMPORARY STORAGE FOR X REGISTER



;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;         This test routine will set up the APPLE II console as         ;
;       device 1 and an APPLE II printer which is assumed to be         ;
;       in slot 1 as device 2. The test routine will then read          ;
;       a line from the console and echo it to the console and          ;
;       the printer.                                                    ;
;                                                                       ;
;                                                                       ;

;EQUATE
CR      .EQU    08DH                    ;APPLE II CARRIAGE RETURN CHARACTER
CBUF    .EQU    0D6H                    ;STARTING ADDRESS OF I/O BUFFER

SC1006:
        ;INITIALIZE DEVICE LIST
        JSR     INITDL

        ;SET UP APPLE CONSOLE AS DEVICE 1
        LDA     CONDVA+1
        LDY     CONDVA
        JSR     ADDDL                   ;ADD CONSOLE DEVICE TO DEVICE LIST
        LDA     #INIT                   ;INITIALIZE OPERATION
        STA     IOCB+IOCBOP
        LDA     #1
        STA     IOCB+IOCBDN             ;DEVICE NUMBER = 1
        LDA     AIOCB+1
        LDY     AIOCB
        JSR     IOHDLR                  ;PERFORM INITIALIZATION
```

```
        ;SET UP APPLE PRINTER AS DEVICE 2
        LDA     PRTDVA+1
        LDY     PRTDVA
        JSR     ADDDL           ;ADD PRINTER DEVICE TO DEVICE LIST
        LDA     #INIT           ;INITIALIZE OPERATION
        STA     IOCB+IOCBOP
        LDA     #2
        STA     IOCB+IOCBDN     ;DEVICE NUMBER = 2
        LDA     AIOCB+1
        LDY     AIOCB
        JSR     IOHDLR          ;INITIALIZE PRINTER DEVICE

        ;LOOP READING LINES FROM CONSOLE, AND ECHOING THEM TO
        ; THE CONSOLE AND PRINTER UNTIL A BLANK LINE IS ENTERED
TSTLP:
        LDA     #1              ;SET DEVICE TO NUMBER 1 (CONSOLE)
        STA     IOCB+IOCBDN
        LDA     #RNBYTE         ;SET OPERATION TO READ N BYTES
        STA     IOCB+IOCBOP
        LDA     #LENBUF         ;SET BUFFER LENGTH TO LENBUF
        STA     IOCB+IOCBBL
        LDA     #0              ;THE HIGH BYTE OF LENBUF IS 0 IN OUR EXAMPLE
        STA     IOCB+IOCBBL+1
        LDA     AIOCB+1         ;SET REGISTERS A,Y TO THE IOCB ADDRESS
        LDY     AIOCB
        JSR     IOHDLR          ;READ A LINE

        ;ECHO THE LINE TO THE CONSOLE
        ;DEVICE IS STILL CONSOLE FROM THE READ LINE ABOVE
        LDA     #WNBYTE         ;SET OPERATION TO WRITE N BYTES
        STA     IOCB+IOCBOP
        LDA     AIOCB+1         ;SET REGISTERS A,Y TO THE IOCB ADDRESS
        LDY     AIOCB
        JSR     IOHDLR          ;WRITE N BYTES

        ;OUTPUT A CARRIAGE RETURN TO CONSOLE
        LDX     #CR             ;SET REGISTER X TO CARRIAGE RETURN CHARACTER
        LDA     #W1BYTE         ;SET OPERATION TO WRITE 1 BYTE
        STA     IOCB+IOCBOP
        LDA     AIOCB+1         ;SET REGISTERS A,Y TO THE IOCB ADDRESS
        LDY     AIOCB
        JSR     IOHDLR          ;WRITE 1 BYTE

        ;ECHO THE LINE TO THE PRINTER ALSO
        LDA     #2              ;SET DEVICE TO NUMBER 2 (PRINTER)
        STA     IOCB+IOCBDN
        LDA     #WNBYTE         ;SET OPERATION TO WRITE N BYTES
        STA     IOCB+IOCBOP
        LDA     AIOCB+1         ;SET REGISTERS A,Y TO THE IOCB ADDRESS
        LDY     AIOCB
        JSR     IOHDLR          ;WRITE N BYTES

        ;WRITE A CARRIAGE RETURN TO THE PRINTER
        LDX     #8DH            ;SET REGISTER X TO CARRIAGE RETURN CHARACTER
        LDA     #W1BYTE         ;SET OPERATION TO WRITE 1 BYTE
```

```
        STA     IOCB+IOCBOP
        LDA     AIOCB+1         ;SET REGISTERS A,Y TO THE IOCB ADDRESS
        LDY     AIOCB
        JSR     IOHDLR          ;WRITE 1 BYTE

        LDA     IOCB+IOCBBL     ;GET LOW BYTE
        LDY     #1
        ORA     IOCB+IOCBBL,Y   ;OR WITH HIGH BYTE
        BNE     TSTLP           ;BRANCH IF BUFFER LENGTH IS NOT ZERO

        BRK
        JMP     SC1006

;IOCB FOR PREFORMING THE IO
AIOCB:  .WORD   IOCB            ;ADDRESS OF THE IOCB
IOCB    .BLOCK  1               ;DEVICE NUMBER
        .BLOCK  1               ;OPERATION NUMBER
        .BLOCK  1               ;STATUS
        .WORD   BUFFER          ;BUFFER ADDRESS
        .WORD   LENBUF          ;BUFFER LENGTH

;BUFFER
LENBUF  .EQU    127
BUFFER  .BLOCK  LENBUF

;DEVICE TABLE ENTRIES
CONDVA: .WORD   CONDV           ;CONSOLE DEVICE ADDRESS
CONDV:  .WORD   0               ;LINK FIELD
        .BYTE   1               ;DEVICE 1
        .WORD   CINIT           ;CONSOLE INITIALIZE
        .WORD   CISTAT          ;CONSOLE INPUT STATUS
        .WORD   CIN             ;CONSOLE INPUT 1 BYTE
        .WORD   CINN            ;CONSOLE INPUT N BYTES
        .WORD   COSTAT          ;CONSOLE OUTPUT STATUS
        .WORD   COUT            ;CONSOLE OUTPUT 1 BYTE
        .WORD   COUTN           ;CONSOLE OUTPUT N BYTES

PRTDVA: .WORD   PRTDV           ;PRINTER DEVICE ADDRESS
PRTDV:  .WORD   0               ;LINK FIELD
        .BYTE   2               ;DEVICE 2
        .WORD   PINIT           ;PRINTER INITIALIZE
        .WORD   0               ;NO PRINTER INPUT STATUS
        .WORD   0               ;NO PRINTER INPUT 1 BYTE
        .WORD   0               ;NO PRINTER INPUT N BYTES
        .WORD   POSTAT          ;PRINTER OUTPUT STATUS
        .WORD   POUT            ;PRINTER OUTPUT 1 BYTE
        .WORD   POUTN           ;PRINTER OUTPUT N BYTES

;************************************
;CONSOLE I/O ROUTINES
;************************************
```

```
;CONSOLE INITIALIZE
CINIT:
        LDA     #0              ;A = STATUS NO ERRORS
        RTS                     ;NO INITIALIZATION NECESSARY

;CONSOLE INPUT STATUS (READY IS BIT 7 OF ADDRESS 0C000H)
CISTAT:
        LDA     0C000H          ;GET KEYBOARD STATUS BYTE
        BPL     CNONE           ;BRANCH IF CHARACTER IS NOT READY
        LDA     #2              ;INDICATE CHARACTER IS READY
        BNE     CIS1            ;BRANCH ALWAYS TAKEN

CNONE:
        LDA     #0              ;NOT READY
CIS1
        LDY     #IOCBST
        STA     (IOCBA),Y       ;STORE STATUS AND LEAVE IT IN REGISTER A
        RTS


;CONSOLE READ 1 BYTE
CIN:
        LDA     C000H
        BPL     CIN             ;WAIT FOR CHARACTER TO BECOME READY
        TXA                     ;MOVE CHARACTER TO REGISTER X
        LDA     #0              ;STATUS = NO ERRORS
        RTS

;CONSOLE READ N BYTES
CINN:
        ;READ LINE USING THE APPLE MONITOR GETLN ROUTINE AT 0FD6AH
        ; 33H = PROMPT LOCATION
        ; 200H = BUFFER ADDRESS
        LDA     #"?" OR 80H     ;SET BIT 7
        STA     033H            ;SET UP APPLE PROMPT CHARACTER
        JSR     0FD6AH          ;CALL APPLE MONITOR GETLN ROUTINE

        ;VERIFY THAT THE NUMBER OF BYTES READ WILL FIT INTO THE CALLERS BUFFER
        LDY     #IOCBBL+1
        LDA     (IOCBA),Y       ;GET HIGH BYTE
        BNE     CINN1           ;BRANCH IF HIGH BYTE IS NOT ZERO
        DEY
        TXA
        CMP     (IOCBA),Y
        BCC     CINN1           ;BRANCH IF THE NUMBER OF CHARACTERS READ IS
                                ; LESS THAN THE BUFFER LENGTH
        BEQ     CINN1           ;BRANCH IF THE LENGTHS ARE EQUAL

        LDA     (IOCBA),Y
        TAX                     ;OTHERWISE TRUNCATE THE NUMBER OF CHARACTERS
                                ; READ TO THE BUFFER LENGTH
CINN1:
        TXA
        STA     (IOCBA),Y       ;SET BUFFER LENGTH TO NUMBER OF CHARACTERS READ
        LDA     #0
        INY
        STA     (IOCBA),Y       ;ZERO UPPER BYTE OF BUFFER LENGTH
```

```
        ;MOVE THE DATA FROM APPLE BUFFER AT 200H TO CALLER'S BUFFER
        LDY     #IOCBBA             ;GET POINTER TO CALLER'S BUFFER FROM IOCB
        LDA     (IOCBA),Y
        STA     CBUF                ;SAVE POINTER ON PAGE ZERO
        INY
        LDA     (IOCBA),Y           ;SET UP MSB OF POINTER ALSO
        STA     CBUF+1
        TXA
        BEQ     CINN3               ;EXIT IF NO BYTES TO MOVE
        LDY     #0

        ;NOW MOVE THE DATA TO CALLER'S BUFFER
CINN2:
        LDA     200H,Y              ;GET A BYTE FROM APPLE BUFFER
        STA     (CBUF),Y            ;MOVE BYTE TO CALLER'S BUFFER
        INY
        DEX
        BNE     CINN2               ;COUNT BYTES

        ;GOOD STATUS (0) - NO ERRORS
CINN3:
        LDA     #0                  ;NO ERRORS
        RTS


;CONSOLE OUTPUT STATUS
COSTAT:
        LDA     #3                  ;STATUS IS ALWAYS READY TO OUTPUT
        RTS


;CONSOLE OUTPUT 1 BYTE
COUT:
        TXA
COUT1:
        JSR     0FDEDH              ;APPLE CHARACTER OUTPUT ROUTINE
        LDA     #0                  ;STATUS = NO ERRORS
        RTS

COUT1A: .WORD   COUT1               ;ADDRESS OF OUTPUT ROUTINE TO BE PLACED IN A,Y


;CONSOLE OUTPUT N BYTES
COUTN:
        LDA     COUT1A+1
        LDY     COUT1A              ;A,Y = ADDRESS OF OUTPUT ROUTINE
        JSR     OUTN                ;CALL OUTPUT N CHARACTERS
        LDA     #0                  ;STATUS = NO ERRORS
        RTS


;****************************************
;PRINTER ROUTINES
; ASSUME PRINTER CARD IS IN SLOT 1
;****************************************
```

```
;PRINTER INITIALIZE
PINIT:
        LDA     #0              ;NOTHING TO DO, RETURN NO ERRORS
        RTS

;PRINTER OUTPUT STATUS
POSTAT:
        LDA     #0              ;ASSUME IT IS ALWAYS READY
        RTS

;PRINTER OUTPUT 1 BYTE
POUT:
        TXA
POUT1:
        JSR     0C107H          ;CHARACTER OUTPUT ROUTINE
        LDA     #0
        RTS

POUT1A: .WORD   POUT1           ;ADDRESS OF CHARACTER OUTPUT ROUTINE TO BE
                                ; PLACED IN A,Y

;PRINTER OUTPUT N BYTES
POUTN:
        LDA     POUT1A+1
        LDY     POUT1A          ;A,Y = ADDRESS OF OUTPUT ROUTINE
        JSR     OUTN            ;CALL OUTPUT N CHARACTERS
        LDA     #0              ;NO ERRORS
        RTS

;********************************************
;ROUTINE: OUTN
;PURPOSE: OUTPUT N CHARACTERS
;ENTRY: REGISTER A = HIGH BYTE OF CHARACTER OUTPUT SUBROUTINE ADDRESS
;       REGISTER Y = LOW BYTE OF CHARACTER OUTPUT SUBROUTINE ADDRESS
;       IOCBA = STARTING ADDRESS OF AN IOCB
;EXIT:  DATA OUTPUT
;REGISTERS USED: ALL
;********************************************

OUTN:
        ;STORE ADDRESS OF THE CHARACTER OUTPUT SUBROUTINE
        STA     COSR+1
        STY     COSR

        ;GET OUTPUT BUFFER ADDRESS FROM IOCB, SAVE ON PAGE ZERO
        LDY     #IOCBBA
        LDA     (IOCBA),Y
        STA     CBUF
        INY
        LDA     (IOCBA),Y
        STA     CBUF+1
```

```
        ;GET BUFFER LENGTH FROM IOCB, EXIT IF IT IS ZERO
        LDY     #IOCBBL
        LDA     (IOCBA),Y
        STA     BUFLEN
        INY
        LDA     (IOCBA),Y
        STA     BUFLEN+1
        ORA     BUFLEN
        BEQ     OUT3            ;BRANCH IF BUFFER LENGTH IS ZERO

        ;START AT BEGINNING OF BUFFER
        LDA     #0
        STA     IDX
OUTLP:
        LDY     IDX
        LDA     (CBUF),Y        ;GET NEXT CHARACTER FROM BUFFER
        JSR     LP0             ;WRITE CHARACTER TO OUTPUT DEVICE
        JMP     LP1

LP0:    JMP     (COSR)          ;OUTPUT THE CHARACTER VIA THE CURRENT
                                ; OUTPUT SUBROUTINE

LP1:
        ;INCREMENT TO THE NEXT CHARACTER IN THE BUFFER
        INC     IDX
        BNE     LP2
        INC     CBUF+1          ;INCREMENT THE HIGH BYTE IS NECESSARY

        ;DECREMENT BUFFER LENGTH, CONTINUE LOOPING IF IT IS NOT ZERO
LP2:
        LDA     BUFLEN
        BNE     DECLS
        DEC     BUFLEN+1        ;BORROW FROM HIGH BYTE IF NECESSARY
DECLS:  DEC     BUFLEN          ;ALWAYS DECREMENT LOW BYTE
        BNE     OUTLP
        LDA     BUFLEN+1
        BNE     OUTLP           ;CONTINUE UNLESS ALL CHARACTERS SENT

OUT3:
        RTS

COSR:   .WORD   0               ;ADDRESS OF THE CHARACTER OUTPUT SUBROUTINE
BUFLEN: .WORD   0               ;TEMPORARY BUFFER LENGTH
IDX:    .BYTE   0               ;TEMPORARY INDEX

        .END
```

Initializes a set of I/O ports from an array of port addresses and initial values. Examples are given of initializing programmable I/O devices such as the 6520 Peripheral Interface Device (Adapter), the 6522 Versatile Interface Adapter, the 6530 Multifunction Device, the 6532 Multifunction Device, the 6551 Asychronous Communications Device Adapter, and the 6850 Asynchronous Communications Device Adapter.

This subroutine is intended as a generalized method for initializing I/O sections. The initialization may involve data ports, data direction registers that determine whether bits are inputs or outputs, control or command registers that determine the operating modes of programmable devices, counters (in timers), priority registers, and other external registers or storage locations.

Some of the tasks the user may perform with this routine are:

1. Assign bidirectional I/O lines as inputs or outputs.

2. Initialize output ports to known starting values.

3. Enable or disable interrupts from peripheral chips.

4. Determine operating modes, such as whether inputs are latched, whether strobes are produced, how priorities are assigned, whether timers operate continuously or only on demand, etc.

5. Load initial counts into timers.

Registers Used: All

Execution Time: 16 cycles overhead plus 52 cycles per port entry. If, for example, NUMBER OF PORT ENTRIES = 10, execution time is

$52 * 10 + 16 = 520 + 16 = 536$ cycles.

Program Size: 40 bytes plus the size of the table (three bytes per entry)

Data Memory Required: Four bytes on page 0, two for a pointer to the array (starting at address ARYADR, $00D0_{16}$ in the listing) and two for a pointer to the port (starting at address PRTADR, $00D2_{16}$ in the listing).

6. Select bit rates for communications.

7. Clear or reset devices that are not tied to the overall system reset line.

8. Initialize priority registers or assign initial priorities to interrupts or other operations.

9. Initialize vectors used in servicing interrupts, DMA requests, and other inputs.

Procedure: The program loops through the specified number of ports, obtaining the port address and the initial value from the array and storing the initial value in the port address. This procedure does not depend on the type of devices used in the I/O section or on the number of devices. Additions and deletions can be made by means of appropriate changes in the array and in the parameters of the routine, without changing the routine itself.

Entry Conditions

(A) = More significant byte of starting address of array of ports and initial values
(Y) = Less significant byte of starting address of array of ports and initial values
(X) = Number of entries in array (number of ports to initialize).

Exit Conditions

All ports initialized.

Example

Data: Number of ports to initialize = 3
Array elements are:
- High byte of port 1 address
- Low byte of port 1 address
- Initial value for port 1
- High byte of port 2 address
- Low byte of port 2 address
- Initial value for port 2
- High byte of port 3 address
- Low byte of port 3 address
- Initial value for port 3

Result: Initial value for port 1 stored in port 1 address
Initial value for port 2 stored in port 2 address
Initial value for port 3 stored in port 3 address.

Note that each element in the array consists of 3 bytes containing:

Less significant byte of port address
More significant byte of port address
Initial value for port

```
;                                                                ;
;                                                                ;
;                                                                ;
;                                                                ;
;       Title           Initialize I/O ports                     ;
;       Name:           IPORTS                                   ;
;                                                                ;
;                                                                ;

;                                                                ;
;       Purpose:        Initialize I/O ports from an array of port ;
;                       addresses and values.                    ;
;                                                                ;
;       Entry:          Register A = High byte of array address  ;
```

```
;                          Register Y = Low byte of array address        ;
;                          Register X = Number of ports to initialize    ;
;                                                                        ;
;                          The array consists of 3 byte elements.        ;
;                             array+0 = High byte of port 1 address      ;
;                             array+1 = Low byte of port 1 address       ;
;                             array+2 = Value to store in port 1 address ;
;                             array+3 = High byte of port 2 address      ;
;                             array+4 = Low byte of port 2 address       ;
;                             array+5 = Value to store in port 2 address ;
;                                        .                               ;
;                                        .                               ;
;                                        .                               ;
;                                                                        ;
;       Exit:              None                                          ;
;                                                                        ;
;       Registers used: All                                              ;
;                                                                        ;
;       Time:              16 cycles overhead plus                       ;
;                          52 cycles per port to initialize              ;
;                                                                        ;
;       Size:              Program 40 bytes                              ;
;                          Data     2 bytes in page zero                 ;
;                                                                        ;
;                                                                        ;


;PAGE ZERO POINTERS
ARYADR  .EQU    0D0H                ;ARRAY ADDRESS
PRTADR  .EQU    0D2H                ;PORT ADDRESS

IPORTS:
        ;SAVE STARTING ADDRESS OF INITIALIZATION ARRAY
        STA     ARYADR+1
        STY     ARYADR

        ;EXIT IF THE NUMBER OF PORTS IS ZERO
        TXA                         ;SET FLAGS
        BEQ     EXITIP              ;EXIT IF NUMBER OF PORTS = 0
        ;LOOP PICKING UP THE PORT ADDRESS AND
        ; SENDING THE VALUE UNTIL ALL PORTS ARE INITIALIZED
LOOP:
        ;GET PORT ADDRESS FROM ARRAY AND SAVE IT
        LDY     #0
        LDA     (ARYADR),Y          ;GET LOW BYTE OF PORT ADDRESS
        STA     PRTADR
        INY
        LDA     (ARYADR),Y          ;GET HIGH BYTE OF PORT ADDRESS
        STA     PRTADR+1

        ;GET THE INITIAL VALUE AND SEND IT TO THE PORT
        INY
        LDA     (ARYADR),Y          ;GET INITIAL VALUE
        LDY     #0
        STA     (PRTADR),Y          ;OUTPUT TO PORT
```

```
        ;POINT TO THE NEXT ARRAY ELEMENT
        LDA     ARYADR
        CLC
        ADC     #3              ;ADD 3 TO LOW BYTE OF THE ADDRESS
        STA     ARYADR
        BCC     LOOP1
        INC     ARYADR+1        ;INCREMENT HIGH BYTE IF A CARRY
LOOP1:

        ;DECREMENT NUMBER OF PORTS TO DO,EXIT WHEN ALL PORTS ARE INITIALIZED
        DEX
        BNE     LOOP

EXITIP:
        RTS


;                                                                  ;
;                                                                  ;
;       SAMPLE EXECUTION:                                          ;
;                                                                  ;
;                                                                  ;


;
;INITIALIZE
;  6520 PIA
;  6522 VIA
;  6530 ROM/RAM/IO/TIMER
;  6532 RAM/IO/TIMER
;  6850 SERIAL INTERFACE(ACIA)
;  6551 SERIAL INTERFACE(ACIA)
SC1007:
        LDA     ADRARY+1
        LDY     ADRARY
        LDX     SZARY
        JSR     IPORTS          ;INITIALIZE THE PORTS
        BRK

ARRAY:
        ;INITIALIZE 6520, ASSUME BASE ADDRESS FOR REGISTERS AT 2000H
        ;  PORT A = INPUT
        ;  CA1 = DATA AVAILABLE, SET ON LOW TO HIGH TRANSITION, NO INTERRUPTS
        ;  CA2 = DATA ACKNOWLEDGE HANDSHAKE
        .WORD   2001H           ;6520 CONTROL REGISTER A ADDRESS
        .BYTE   00000000B       ;INDICATE NEXT ACCESS TO DATA DIRECTION
                                ; REGISTER (SAME ADDRESS AS DATA REGISTER)
        .WORD   2000H           ;6520 DATA REGISTER A ADDRESS
        .BYTE   00000000B       ;ALL BITS = INPUT
        .WORD   2001H           ;6520 CONTROL REGISTER A ADDRESS
        .BYTE   00100110B       ;SET UP CA1,CA2 AND SET BIT 2 TO DATA REGISTER

        ;  PORT B = OUTPUT
        ;  CB1 = DATA ACKNOWLEDGE,SET ON HIGH TO LOW TRANSITION,NO INTERRUPTS
        ;  CB2 = DATA AVAILABLE, CLEARED BY WRITING DATA REGISTER B
        ;                        SET TO 1 BY HIGH TO LOW TRANSITION ON CB1
```

```
.WORD   2003H           ;6520 CONTROL REGISTER B ADDRESS
.BYTE   00000000B       ;INDICATE NEXT ACCESS TO DATA DIRECTION
                        ; REGISTER
.WORD   2002H           ;6520 DATA REGISTER B ADDRESS
.BYTE   11111111B       ;ALL BITS = OUTPUT
.WORD   2003H           ;6520 CONTROL REGISTER B ADDRESS
.BYTE   00100100B       ;SET UP CB1,CB2 AND SET BIT 2 TO DATA REGISTER

;INITIALIZE 6522, ASSUME BASE ADDRESS FOR REGISTERS AT 2010H
; PORT A = BITS 0..3 = OUTPUT, BITS 4..7 = INPUT
; CA1, CA2 ARE NOT USED.
; PORT B = LATCHED INPUT
;  CB1 = DATA AVAILABLE, SET ON LOW TO HIGH TRANSITION
;  CB2 = DATA ACKNOWLEDGE HANDSHAKE
.WORD   2013H           ;6522 DATA DIRECTION REGISTER A
.BYTE   00001111B       ;BITS 0..3 = OUTPUT, 4..7 = INPUT
.WORD   2012H           ;6522 DATA DIRECTION REGISTER B
.BYTE   00000000B       ;ALL BITS = INPUT
.WORD   201CH           ;6522 PERIPHERAL CONTROL REGISTER
.BYTE   10010000B       ;SET UP CB1, CB2
.WORD   201BH           ;6522 AUXILIARY CONTROL REGISTER
.BYTE   00000010B       ;MAKE PORT B LATCH THE INPUT DATA

;INITIALIZE 6530, ASSUME BASE ADDRESS FOR REGISTERS AT 2020H
; PORT A = OUTPUT
; PORT B = INPUT
.WORD   2021H           ;6530 DATA DIRECTION REGISTER A
.BYTE   11111111B       ;ALL BITS = OUTPUT
.WORD   2023H           ;6530 DATA DIRECTION REGISTER B
.BYTE   00000000B       ;ALL BITS = INPUT

;INITIALIZE 6532, ASSUME BASE ADDRESS FOR REGISTERS AT 2030H
; PORT A = BITS 0..6 = OUTPUT
;          BIT 7 = INPUT FOR PORT B DATA AVAILABLE.
; PORT B = INPUT
.WORD   2031H           ;6532 DATA DIRECTION REGISTER A
.BYTE   01111111B       ;BITS 0..6 = OUTPUT, BIT 7 = INPUT
.WORD   2033H           ;6532 DATA DIRECTION REGISTER B
.BYTE   00000000B       ;ALL BITS = INPUT

;INITIALIZE 6551, ASSUME BASE ADDRESS FOR REGISTERS AT 2040H
; 8 BIT DATA, NO PARITY
; 1 STOP BIT
; 9600 BAUD FROM ON BOARD BAUD RATE GENERATOR
; NO INTERRUPTS
.WORD   2041H           ;WRITE TO 6551 STATUS REGISTER TO RESET
.BYTE   0               ;THIS VALUE COULD BE ANYTHING
.WORD   2042H           ;6551 CONTROL REGISTER
.BYTE   10011110B       ;1 STOP, 8 BIT DATA, INTERNAL 9600 BAUD
.WORD   2043H           ;6551 COMMAND REGISTER
.BYTE   00000011B       ;NO PARITY, NO ECHO, NO RECEIVER INTERRUPT,
                        ;DTR LOW

;INITIALIZE 6850, ASSUME BASE ADDRESS FOR REGISTERS AT 2050H
; 8 BIT DATA, NO PARITY
```

```
        ; 1 STOP BIT
        ; DIVIDE MASTER CLOCK BY 1
        ; NO INTERRUPTS
        .WORD   2050H               ;WRITE TO 6850 CONTROL REGISTER
        .BYTE   00000011B           ;PERFORM A MASTER RESET
        .WORD   2050H               ;6850 CONTROL REGISTER
        .BYTE   00010101B           ;NO INTERRUPTS, RTS LOW,
                                    ;8 BITS, 1 STOP, DIVIDE BY 1

ENDARY:                                     ;END OF ARRAY
ADRARY: .WORD   ARRAY                       ;ADDRESS OF ARRAY
SZARY:  .BYTE   (ENDARY - ARRAY) / 3        ;NUMBER OF PORTS TO INITIALIZE

        .END    ;PROGRAM
```

Provides a delay of between 1 and 255 milliseconds, depending on the parameter supplied. The user must calculate the value MSCNT to fit a particular computer.

MSCENT = (100/CYCLETIME − 10)/5
= 200/CYCLETIME − 2

CYCLETIME is the number of microseconds per clock period for a particular computer (1 for KIM-1, SYM-1, and AIM-65, 0.9799269 for APPLE II™).

Procedure: The program simply counts down the index registers for the appropriate amount of time as determined by the user-supplied constant. A few extra NOPs take account of the call instruction, the return instruction, and the routine overhead.

Registers Used: X, Y, P

Execution Time: 1 millisecond * (Y). If (Y) = 0, the minimum time is 17 cycles including a JSR instruction.

Program Size: 156 bytes

Data Memory Required: None

Special Case: (Y) = 0 causes an exit with a minimum execution time of 17 cycles including a JSR instruction. (Y) = 0 and (X) is unchanged.

Entry Conditions

(Y) = Number of milliseconds to delay (1 to 255).

Exit Conditions

Returns after the specified number of milliseconds with (X) = (Y) = 0.

Example

Data: (Y) = number of milliseconds = $2A_{16} = 42_{10}$

Result: Software delay of $2A_{16}$ (42_{10}) milliseconds, assuming that user supplies the proper value of MSCNT.

```
;                                                             ;
;                                                             ;
;                                                             ;
;                                                             ;
;       Title          Delay milliseconds                     ;
;       Name:          Delay                                  ;
;                                                             ;
;                                                             ;

;                                                             ;
;       Purpose:       Delay from 1 to 255 milliseconds       ;
```

```
;                                                                          ;
;       Entry:          Register Y = number of milliseconds to delay.      ;
;                                                                          ;
;       Exit:           Returns to calling routine after the               ;
;                       specified delay.                                   ;
;                                                                          ;
;       Registers used: X,Y,P                                              ;
;                                                                          ;
;       Time:           1 millisecond * Register Y.                        ;
;                       If Y = 0 then the minimum time is 17               ;
;                       cycles including the JSR overhead.                 ;
;                                                                          ;
;                                                                          ;
;       Size:           Program 29 bytes                                   ;
;                       Data      NONE                                     ;
;                                                                          ;
;                                                                          ;


; HERE IS THE FORMULA FOR COMPUTING THE DELAY COUNTS MSCNT1 AND MSCNT2
;
; MSCNT = 200/CYCLETIME - 2 WHERE CYCLE TIME IS THE LENGTH
;        OF A PARTICULAR COMPUTER'S CLOCK PERIOD IN MICROSECONDS
;
; EXAMPLES: KIM, SYM, AIM HAVE 1 MHz CLOCKS, SO MSCNT = 198,
;           APPLE HAS A 1.023 MHz CLOCK, SO MSCNT = 202.
;
; IN THE LAST ITERATION, WE REDUCE THE COUNT BY 3 (MSCNT)
;  TO DELAY 1 MILLISECOND LESS THE OVERHEAD WHERE THE
;  OVERHEAD IS:
;                6 CYCLES ==> JSR DELAY
;                2 CYCLES ==> CPY #0
;                2 CYCLES ==> BEQ EXIT (ASSUMED NOT TAKEN)
;                2 CYCLES ==> NOP
;                2 CYCLES ==> CPY #1
;                3 CYCLES ==> BNE DELAYA (ASSUMED TAKEN)
;                2 CYCLES ==> DEY
;               -1 CYCLE  ==> THE LAST BNE DELAY1 NOT TAKEN
;                2 CYCLES ==> LDX #MSCNT2
;               -1 CYCLE  ==> THE LAST BNE DELAY2 NOT TAKEN
;                6 CYCLES ==> RTS
;             ------
;               25 CYCLES OVERHEAD
;
;EQUATES
;       1 MHZ CLOCK
;MSCNT  .EQU    0C6H    ;198 TIMES THROUGH DELAY1
;
;       APPLE (1.023 MHZ)
MSCNT   .EQU    0CAH    ;202 TIMES THROUGH DELAY1

DELAY:
        CPY     #0      ; 2 CYCLES
        BEQ     EXIT    ; 2 CYCLES (EXIT IF DELAY = 0)
        NOP             ; 2 CYCLES (TO MAKE OVERHEAD = 25 CYCLES)
```

```
        ;IF DELAY IS TO BE 1 MILLISECOND THEN GOTO LAST1
        ; THIS LOGIC IS DESIGNED TO BE 5 CYCLES THROUGH EITHER PATH
        CPY     #1      ; 2 CYCLES
        BNE     DELAYA  ; 3 CYCLES (IF TAKEN ELSE 2 CYCLES)
        JMP     LAST1   ; 3 CYCLES

        ;DELAY 1 MILLISECOND TIMES(Y-1)
DELAYA:
        DEY             ; 2 CYCLES  (PREDECREMENT Y)
DELAY0:
        LDX     #MSCNT  ; 2 CYCLES
DELAY1:
        DEX             ; 2 CYCLES
        BNE     DELAY1  ; 3 CYCLES
        NOP             ; 2 CYCLES
        NOP             ; 2 CYCLES
        DEY             ; 2 CYCLES
        BNE     DELAY0  ; 3 CYCLES

LAST1:
        ;DELAY THE LAST TIME 25 CYCLES LESS TO TAKE THE
        ; CALL, RETURN, AND ROUTINE OVERHEAD INTO ACCOUNT
        LDX     #MSCNT-3 ; 2 CYCLES
DELAY2:
        DEX             ; 2 CYCLES
        BNE     DELAY2  ; 3 CYCLES

EXIT:
        RTS             ; 6 CYCLES



;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION:                                            ;
;                                                                    ;
;                                                                    ;


SC1008:
        ;
        ;DELAY 10 SECONDS
        ;  CALL DELAY 40 TIMES AT 250 MILLISECONDS EACH
        LDA     #40     ;40 TIMES (28 HEX)
        STA     COUNT

        ;DELAY 1/4 SECOND
QTRSCD:
        LDY     #250    ;250 MILLISECONDS (FA HEX)
        JSR     DELAY
        DEC     COUNT
        BNE     QTRSCD

        BRK             ;STOP AFTER 10 SECONDS
        JMP     SC1008
```

```
;DATA SECTION
COUNT    .BYTE    0

         .END     ;PROGRAM
```

Unbuffered Interrupt-Driven Input/Output Using a 6850 ACIA (SINTIO)

Performs interrupt-driven input and output using a 6850 ACIA and single-character input and output buffers. Consists of the following subroutines:

1. INCH reads a character from the input buffer.

2. INST determines whether there is a character available in the input buffer.

3. OUTCH writes a character into the output buffer.

4. OUTST determines whether the output buffer is full.

5. INIT initializes the 6850 ACIA, the interrupt vectors, and the software flags (used to transfer data between the main program and the interrupt service routine).

6. IOSRVC determines which interrupt occurred and provides the proper input or output service. In response to the input interrupt, it reads a character from the ACIA into the input buffer. In response to the output interrupt, it writes a character from the output buffer into the ACIA.

Examples describe a 6850 ACIA on an Apple II serial I/O board in slot 1.

Procedures:

1. INCH waits for a character to become available, clears the Data Ready flag (RECDF), and loads the character into the accumulator.

2. INST sets the Carry flag from the Data Ready flag (memory location RECDF).

3. OUTCH waits for the character buffer to empty, places the character in the buffer, and sets the Character Available flag (TRNDF).

Registers Used:

1. INCH A, F, Y
2. INST A, F
3. OUTCH A, F, Y
4. OUTST A, F
5. INIT A, F

Execution Time:

1. INCH 33 cycles if a character is available
2. INST 12 cycles
3. OUTCH 92 cycles if the output buffer is empty and the ACIA is ready to send data
4. OUTST 12 cycles
5. INIT 73 cycles
6. IOSRVC 39 cycles to service an input interrupt, 59 cycles to service an output interrupt, 24 cycles to determine interrupt is from another device

Program Size: 168 bytes

Data Memory Required: Six bytes anywhere in RAM. One byte for the received data (at address RECDAT), one byte for the receive data flag (at address RECDF), one byte for the transmit data (at address TRNDAT), one byte for the transmit data flag (at address TRNDF), and two bytes for the address of the next interrupt service routine (starting at address NEXTSR).

4. OUTST sets the Carry flag from the Character Available flag (memory location TRNDF).

5. INIT clears the software flags, sets up the interrupt vector, resets the ACIA (a master reset, since the ACIA has no reset input), and initializes the ACIA by placing the appropriate value in its control register (input interrupts enabled, output interrupts disabled).

6. IOSRVC determines whether the interrupt was an input interrupt (bit 0 of the ACIA status register = 1), an output interrupt (bit

1 of the ACIA status register = 1), or the product of some other device. If the input interrupt occurred, the program reads the data, saves it in memory, and sets the Data Ready flag (RECDF). If the output interrupt occurred, the program determines whether data is available. If not, the program simply disables the output interrupt. If data is available, the program sends it to the ACIA, clears the Character Available flag (TRNDF), and enables both the input and the output interrupts.

The only special problem in using these routines is that an output interrupt may occur when no data is available. We cannot ignore the interrupt or it will assert itself indefinitely, creating an endless loop. The solution is to disable output interrupts. But now we create a new problem when data is ready to be sent. That is, if we have disabled output interrupts, the system cannot learn from an interrupt that the ACIA is ready to transmit. The solution to this is to create an additional, non-interrupt-driven entry to the routine that sends a character to the ACIA. Since this entry is not caused by an interrupt, we must check the ACIA to see that its output register is actually empty before sending it a character.

The special sequence of operations is the following:

1. Output interrupt occurs before new data is available (that is, the ACIA becomes ready for data). The response is to disable the output interrupt, since there is no data to be sent. Note that this sequence will not occur initially, since INIT disables the output interrupt. Otherwise, the output interrupt would occur immediately, since the ACIA surely starts out empty and therefore ready to transmit data.

2. Output data becomes available. That is, the system now has data to transmit. But there is no use sitting back and waiting for the output interrupt, since it has been disabled.

3. The main program calls the routine (OUTDAT) that sends data to the ACIA. Checking the ACIA's status shows that it is, in fact, ready to transmit a character (it told us it was when the output interrupt occurred). The routine then sends the character and reenables the interrupts.

The basic problem here is that output devices may request service before the computer is ready for them. That is, the devices can accept data but the computer has nothing to send. In particular, we have an initialization problem caused by output interrupts asserting themselves and expecting service. Input devices, on the other hand, request service only when they have data. They start out in the not ready state; that is, an input device has no data to send initially, while the computer is ready to accept data. Thus output devices cause more initialization and sequencing problems in interrupt-driven systems than do input devices.

Our solution may, however, result in an odd situation. Let us assume that the system has some data ready for output but the ACIA is not yet ready for it. Then the system must wait with interrupts disabled for the ACIA to become ready; that is, an interrupt-driven system must disable its interrupts and wait idly, polling the output device. We could eliminate this drawback by keeping a software flag that would be changed when the output interrupt occurred at a time when there was no data. Then the system could check the software flag and determine whether the output interrupt had already occurred. (See Subroutine 11C.)

Entry Conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in accumulator
4. OUTST: none
5. INIT: none

Exit Conditions

1. INCH: character in accumulator
2. INST: Carry flag = 0 if no character is available, 1 is a character is available
3. OUTCH: none
4. OUTST: Carry flag = 0 if output buffer is empty, 1 if it is full.

```
;       Title           Simple interrupt input and output using a 6850  ;
;                       ACIA and a single character buffer.             ;
;       Name:           SINTIO                                          ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Purpose:        This program consists of 5 subroutines which    ;
;                       perform interrupt driven input and output using ;
;                       a 6850 ACIA.                                    ;
;                                                                       ;
;                       INCH                                            ;
;                         Read a character.                             ;
;                       INST                                            ;
;                         Determine input status (whether the input     ;
;                         buffer is empty).                             ;
;                       OUTCH                                           ;
;                         Write a character.                            ;
;                       OUTST                                           ;
;                         Determine output status (whether the output   ;
;                         buffer is full).                              ;
;                       INIT                                            ;
;                         Initialize.                                   ;
;                                                                       ;
;       Entry:          INCH                                            ;
;                         No parameters.                                ;
;                       INST                                            ;
;                         No parameters.                                ;
;                       OUTCH                                           ;
;                         Register A = character to transmit            ;
;                       OUTST                                           ;
;                         No parameters.                                ;
;                       INIT                                            ;
;                         No parameters.                                ;
;                                                                       ;
;       Exit:           INCH                                            ;
;                         Register A = character.                       ;
;                       INST                                            ;
;                         Carry flag equals 0 if input buffer is empty, ;
;                         1 if character is available.                  ;
```

```
;                           OUTCH                                   ;
;                             No parameters                         ;
;                           OUTST                                   ;
;                             Carry flag equals 0 if output buffer is ;
;                             empty, 1 if it is full.               ;
;                           INIT                                    ;
;                             No parameters.                        ;
;                                                                   ;
;       Registers used: INCH                                        ;
;                             A,F,Y                                 ;
;                           INST                                    ;
;                             A,F                                   ;
;                           OUTCH                                   ;
;                             A,F,Y                                 ;
;                           OUTST                                   ;
;                             A,F                                   ;
;                           INIT                                    ;
;                             A,F                                   ;
;                                                                   ;
;       Time:               INCH                                    ;
;                             33 cycles if a character is available ;
;                           INST                                    ;
;                             12 cycles                             ;
;                           OUTCH                                   ;
;                             92 cycles if the output buffer is empty and ;
;                                the ACIA is ready to transmit      ;
;                           OUTST                                   ;
;                             12 cycles                             ;
;                           INIT                                    ;
;                             73 cycles                             ;
;                           IOSRVC                                  ;
;                             24 cycles minimum if the interrupt is not ours;
;                             39 cycles to service a input interrupt ;
;                             59 cycles to service a output interrupt ;
;                                                                   ;
;       Size:               Program 168 bytes                       ;
;                           Data      6 bytes                       ;
;                                                                   ;
;                                                                   ;

;EXAMPLE 6850 ACIA PORT DEFINITIONS FOR AN APPLE SERIAL BOARD IN SLOT 1
ACIASR  .EQU    0C094H          ;ACIA STATUS REGISTER
ACIADR  .EQU    0C095H          ;ACIA DATA REGISTER
ACIACR  .EQU    0C094H          ;ACIA CONTROL REGISTER
IRQVEC  .EQU    03FEH           ;APPLE IRQ VECTOR ADDRESS

;READ A CHARACTER
INCH:
        JSR     INST            ;GET INPUT STATUS
        BCC     INCH            ;WAIT IF CHARACTER IS NOT AVAILABLE
        PHP                     ;SAVE CURRENT STATE OF INTERRUPT SYSTEM
        SEI                     ;DISABLE INTERRUPTS
        LDA     #0
        STA     RECDF           ;INDICATE BUFFER IS NOW EMPTY
        LDA     RECDAT          ;GET THE CHARACTER FROM THE BUFFER
        PLP                     ;RESTORE FLAGS
```

```
        RTS

;RETURN INPUT STATUS (CARRY = 1 IF DATA IS AVAILABLE)
INST:
        LDA     RECDF               ;GET THE DATA READY FLAG
        LSR     A                   ;SET CARRY FROM FLAG
                                    ; CARRY = 1 IF CHARACTER IS AVAILABLE
        RTS
;WRITE A CHARACTER
OUTCH:
        PHP                         ;SAVE STATE OF INTERRUPT FLAG
        PHA                         ;SAVE CHARACTER TO OUTPUT

        ;WAIT FOR THE CHARACTER BUFFER TO EMPTY, THEN STORE THE NEXT CHARACTER
WAITOC:
        JSR     OUTST               ;GET THE OUTPUT STATUS
        BCS     WAITOC              ;WAIT IF THE OUTPUT BUFFER IS FULL
        SEI                         ;DISABLE INTERRUPTS WHILE LOOKING AT THE
                                    ; SOFTWARE FLAGS
        PLA                         ;GET THE CHARACTER
        STA     TRNDAT              ;STORE THE CHARACTER
        LDA     #0FFH               ;INDICATE CHARACTER AVAILABLE (BUFFER FULL)
        STA     TRNDF
        JSR     OUTDAT              ;SEND THE DATA TO THE PORT
        PLP                         ;RESTORE FLAGS
        RTS

;OUTPUT STATUS (CARRY = 1 IF BUFFER IS FULL)
OUTST:
        LDA     TRNDF               ;CARRY = 1 IF CHARACTER IS IN THE BUFFER
        LSR     A
        RTS

;INITIALIZE
INIT:
        PHP                         ;SAVE CURRENT STATE OF FLAGS
        SEI                         ;DISABLE INTERRUPTS DURING INITIALIZATION

        ;INITIALIZE THE SOFTWARE FLAGS
        LDA     #0
        STA     RECDF               ;NO INPUT DATA AVAILABLE
        STA     TRNDF               ;OUTPUT BUFFER EMPTY

        ;SAVE THE CURRENT IRQ VECTOR IN NEXTSR
        LDA     IRQVEC
        STA     NEXTSR
        LDA     IRQVEC+1
        STA     NEXTSR+1

        ;SET THE IRQ VECTOR TO OUR INPUT SERVICE ROUTINE
        LDA     AIOS
        STA     IRQVEC
        LDA     AIOS+1
        STA     IRQVEC+1

        ;INITIALIZE THE 6850
        LDA     #011B
        STA     ACIACR              ;MASTER RESET ACIA
        LDA     #10010001B
        STA     ACIACR              ;INITIALIZE ACIA MODE TO
```

```
                                ;  DIVIDE BY 16
                                ;  8 DATA BITS
                                ;  2 STOP BITS
                                ;  OUTPUT INTERRUPTS DISABLED (NOTE THIS)
                                ;  INPUT INTERRUPTS ENABLED

        PLP                     ;RESTORE CURRENT STATE OF THE FLAGS
        RTS

AIOS:   .WORD   IOSRVC          ;ADDRESS OF INPUT OUTPUT SERVICE ROUTINE

;INPUT OUTPUT INTERRUPT SERVICE ROUTINE
IOSRVC:
        PHA                     ;SAVE REGISTER A
        CLD                     ;BE SURE PROCESSOR IS IN BINARY MODE

        ;GET THE ACIA STATUS: BIT 0 = 1 IF AN INPUT INTERRUPT
        ;BIT 1 = 1 IF AN OUTPUT INTERRUPT
        LDA     ACIASR
        LSR     A               ;BIT 0 TO CARRY
        BCS     IINT            ;BRANCH IF AN INPUT INTERRUPT
        LSR     A               ;BIT 1 TO CARRY
        BCS     OINT            ;BRANCH IF AN OUTPUT INTERRUPT

        ;THE INTERRUPT WAS NOT CAUSED BY THIS ACIA
        PLA
        JMP     (NEXTSR)        ;GOTO THE NEXT SERVICE ROUTINE

;SERVICE INPUT INTERRUPTS
IINT:
        LDA     ACIADR          ;READ THE DATA
        STA     RECDAT          ;STORE IT AWAY
        LDA     #0FFH
        STA     RECDF           ;INDICATE WE HAVE A CHARACTER IN RECDAT
        JMP     EXIT            ;EXIT IOSRVC

;SERVICE OUTPUT INTERRUPTS
OINT:
        LDA     TRNDF           ;GET DATA AVAILABLE FLAG
        BEQ     NODATA          ;BRANCH IF NO DATA TO SEND
        JSR     OUTDT1          ; ELSE OUTPUT THE DATA,
                                ;  (WE DO NOT NEED TO TEST THE STATUS)
        JMP     EXIT

;IF AN OUTPUT INTERRUPT OCCURS WHEN NO DATA IS AVAILABLE,
; WE MUST DISABLE THE INTERRUPT TO AVOID AN ENDLESS LOOP.
; LATER WHEN A CHARACTER BECOMES AVAILABLE, WE CALL THE
; OUTPUT  ROUTINE, OUTDAT, WHICH MUST TEST ACIA STATUS BEFORE
; SENDING THE DATA. THE OUTPUT ROUTINE MUST ALSO REENABLE THE OUTPUT
; INTERRUPT AFTER SENDING THE DATA. THIS PROCEDURE OVERCOMES THE
; PROBLEMS OF AN UNSERVICED OUTPUT INTERRUPT ASSERTING ITSELF
; REPEATEDLY, WHILE STILL ENSURING THAT OUTPUT INTERRUPTS ARE
; RECOGNIZED AND THAT DATA IS NEVER SENT TO AN ACIA THAT IS
; NOT READY FOR IT. THE BASIC PROBLEM HERE IS THAT AN OUTPUT
; DEVICE MAY REQUEST SERVICE BEFORE THE COMPUTER HAS
; ANYTHING TO SEND (WHEREAS AN INPUT DEVICE HAS DATA WHEN IT
```

```
; REQUESTS SERVICE)

NODATA:
        LDA     #10010001B      ;DISABLE OUTPUT INTERRUPTS, ENABLE INPUT
                                ; INTERRUPTS, 8 DATA BITS, 2 STOP BITS, DIVIDE
                                ; BY 16 CLOCK
        STA     ACIACR          ;TURN OFF OUTPUT INTERRUPTS

EXIT:
        PLA                     ;RESTORE REGISTER A
        RTI                     ;RETURN FROM INTERRUPT


;****************************************
;ROUTINE: OUTDAT, OUTDT1 (OUTDAT IS NON-INTERRUPT DRIVEN ENTRY POINT)
;PURPOSE: SEND A CHARACTER TO THE ACIA
;ENTRY: TRNDAT = CHARACTER TO SEND
;EXIT:  NONE
;REGISTERS USED: A,F
;****************************************

;NON-INTERRUPT ENTRY. MUST CHECK IF ACIA IS READY OR WAIT FOR IT
OUTDAT:
        LDA     ACIASR          ;CAME HERE WITH INTERRUPTS DISABLED
        AND     #00000010B      ;TEST THE ACIA OUTPUT REGISTER FOR EMPTY
        BEQ     OUTDAT          ;BRANCH IF IT IS NOT EMPTY

OUTDT1: LDA     TRNDAT          ;GET THE CHARACTER
        STA     ACIADR          ;OUTPUT DATA
        LDA     #0
        STA     TRNDF           ;INDICATE BUFFER EMPTY
        LDA     #10110001B
        STA     ACIACR          ;ENABLE 6850 OUTPUT AND INPUT INTERRUPTS,
                                ; 8 DATA BITS, 2 STOP BITS, DIVIDE BY 16 CLOCK
        RTS


;DATA SECTION
RECDAT  .BLOCK  1               ;RECEIVE DATA
RECDF   .BLOCK  1               ;RECEIVE DATA FLAG (0 = NO DATA, FF = DATA)
TRNDAT  .BLOCK  1               ;TRANSMIT DATA
TRNDF   .BLOCK  1               ;TRANSMIT DATA FLAG (0 = BUFFER EMPTY,
                                ;                    FF = BUFFER FULL)
NEXTSR  .BLOCK  2               ;ADDRESS OF THE NEXT INTERRUPT SERVICE ROUTINE


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC1101:
        JSR     INIT            ;INITIALIZE
        CLI                     ;ENABLE INTERRUPTS
```

```
        ;SIMPLE EXAMPLE
LOOP:
        JSR     INCH            ;READ A CHARACTER
        PHA
        JSR     OUTCH           ;ECHO IT
        PLA
        CMP     #1BH            ;IS IT AN ESCAPE CHARACTER ?
        BNE     LOOP            ;STAY IN LOOP IF NOT
        BRK

        ;AN ASYNCHRONOUS EXAMPLE
        ; OUTPUT "A" TO THE CONSOLE CONTINUOUSLY BUT ALSO LOOK AT THE
        ; INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS.
ASYNLP:
        ;OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        JSR     OUTST           ;IS OUTPUT BUSY ?
        BCS     ASYNLP          ;BRANCH IF IT IS
        LDA     #"A"
        JSR     OUTCH           ;OUTPUT THE CHARACTER

        ;GET A CHARACTER FROM THE INPUT PORT IF ANY
        JSR     INST            ;IS INPUT DATA AVAILABLE ?
        BCC     ASYNLP          ;BRANCH IF NOT (SEND ANOTHER "A")
        JSR     INCH            ;GET THE CHARACTER
        CMP     #1BH            ;IS IT AN ESCAPE CHARACTER ?
        BEQ     DONE            ;BRANCH IF IT IS
        JSR     OUTCH           ;ELSE ECHO IT
        JMP     ASYNLP          ;AND CONTINUE

DONE:
        BRK
        JMP     SC1101

        .END    ;PROGRAM
```

Unbuffered Interrupt-Driven Input/Output Using a 6522 VIA (PINTIO)

Performs interrupt-driven input and output using a 6522 VIA and single-character input and output buffers. Consists of the following subroutines:

1. INCH reads a character from the input buffer.

2. INST determines whether there is a character available in the input buffer.

3. OUTCH writes a character into the output buffer.

4. OUTST determines whether the output buffer is full.

5. INIT initializes the 6522 VIA, the interrupt vectors, and the software flags.

6. IOSRVC determines which interrupt occurred and provides the proper input or output service (i.e., it reads a character from the VIA into the input buffer in response to the input interrupt and it writes a character from the output buffer into the VIA in response to the output interrupt).

Examples describe a 6522 VIA attached to an Apple II computer.

Registers Used:

1. INCH: A, F, Y
2. INST: A, F
3. OUTCH: A, F, Y
4. INIT A, F

Execution Time:

1. INCH: 33 cycles if a character is available
2. INST: 12 cycles
3. OUTCH: 83 cycles if the output buffer is empty and the VIA is ready for data
4. OUTST: 12 cycles
5. INIT: 93 cycles
6. IOSRVC: 43 cycles to service an input interrupt, 81 cycles to service an output interrupt, 24 cycles to determine that interrupt is from another device

Program Size: 194 bytes

Data Memory Required: Seven bytes anywhere in RAM. One byte for the received data (at address RECDAT), one byte for the Receive Data flag (at address RECDF), one byte for the transmit data (at address TRNDAT), one byte for the Transmit Data flag (at address TRNDF), one byte for the Output Interrupt flag (at address OIE), and two bytes for the address of the next interrupt service routine (starting at address NEXTSR).

Procedure:

1. INCH waits for a character to be available in the input buffer, clears the Data Ready flag (RECDF), and loads the character from the buffer into the accumulator.

2. INST sets the Carry flag from the Data Ready flag (memory location RECDF).

3. OUTCH waits for the output buffer to be emptied, places the character (from the accumulator) in the buffer, and sets the character available (buffer full) flag (TRNDF). If an unserviced output interrupt has occurred (i.e., the output device has requested service when no data was available), OUTCH actually sends the data to the VIA.

4. OUTST sets the Carry flag from the Character Available flag (memory location TRNDF).

5. INIT clears the software flags, sets up the interrupt vector, and initializes the 6522 VIA. It makes port A an input port, port B an output port, control lines CA1 and CB1 active low-to-high, control line CA2 a brief

output pulse indicating input acknowledge (active-low after the CPU reads the data), and control line CB2 a write strobe (active-low after the CPU writes the data and lasting until the peripheral becomes ready again). INIT also enables the input interrupt on CA1 and the output interrupt on CB1.

6. IOSRVC determines whether the interrupt was an input interrupt (bit 1 of the VIA interrupt flag register = 1), an output interrupt (bit 4 of the VIA interrupt flag register = 1), or the product of some other device. If the input interrupt occurred, the program reads the data, saves it in the input buffer, and sets the Data Ready flag (RECDF). If the output interrupt occurred, the program determines whether any data is available. If not, the program simply clears the interrupt and clears the flag (OIE) that indicates the output device is actually ready (that is, an output interrupt has occurred at a time when no data was available). If data is available, the program sends it from the output buffer to the VIA, clears the Character Available flag (TRNDF), sets the Output Interrupt flag (OIE), and enables both the input and the output interrupts.

The only special problem in using these routines is that an output interrupt may occur when no data is available to send. We cannot ignore the interrupt or it will assert itself indefinitely, creating an endless loop. The solution is to simply clear the interrupt by reading the data register in port B. But now we create a new problem when the main program has data ready to be sent. The interrupt indicating that the output device is ready has already occurred (and been cleared), so there is no use waiting for it. The solution is to establish an extra flag that indicates (with a 0) that the output interrupt has occurred without being serviced. We call this flag OIE, the Output Interrupt flag. The initialization routine sets it initially (since the output device has not requested service), and the output service routine clears it when an output interrupt occurs that cannot be serviced (no data is available) and sets it after sending data to the VIA (in case it might have been cleared). Now the output routine OUTCH can check OIE to determine whether the output interrupt has already occurred (a 0 value indicates it has, FF hex that it has not).

Note that we can clear a VIA interrupt without actually sending any data. We cannot do this with a 6850 ACIA (see Subroutines 11A and 11C), so the procedures there are somewhat different. This problem of unserviced interrupts occurs only with output devices, since input devices request service only when they have data ready to transfer.

Entry Conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in accumulator
4. OUTST: none
5. INIT: none

Exit Conditions

1. INCH: character in accumulator
2. INST: Carry flag = 0 if no character is available, 1 if a character is available
3. OUTCH: none
4. OUTST: Carry flag = 0 if output buffer is empty, 1 if it is full.
5. INIT: none

```
;       Title               Simple interrupt input and output using a 6522 ;
;                           VIA and a single character buffer.             ;
;       Name:               PINTIO                                         ;
;                                                                          ;
;                                                                          ;

;                                                                          ;
;       Purpose:            This program consists of 5 subroutines which   ;
;                           perform interrupt driven input and output using ;
;                           a 6522 VIA.                                    ;
;                                                                          ;
;                           INCH                                           ;
;                             Read a character.                            ;
;                           INST                                           ;
;                             Determine input status (whether the input    ;
;                             buffer is empty).                            ;
;                           OUTCH                                          ;
;                             Write a character.                           ;
;                           OUTST                                          ;
;                             Determine output status (whether the output  ;
;                             buffer is full).                             ;
;                           INIT                                           ;
;                             Initialize.                                  ;
;                                                                          ;
;       Entry:              INCH                                           ;
;                             No parameters.                               ;
;                           INST                                           ;
;                             No parameters.                               ;
;                           OUTCH                                          ;
;                             Register A = character to transmit           ;
;                           OUTST                                          ;
;                             No parameters.                               ;
;                           INIT                                           ;
;                             No parameters.                               ;
;                                                                          ;
;       Exit:               INCH                                           ;
;                             Register A = character.                      ;
;                           INST                                           ;
```

```
;                            Carry flag equals 0 if input buffer is empty, ;
;                            1 if character is available.                  ;
;                          OUTCH                                           ;
;                            No parameters                                 ;
;                          OUTST                                           ;
;                            Carry flag equals 0 if output buffer is       ;
;                            empty, 1 if it is full.                       ;
;                          INIT                                            ;
;                            No parameters.                                ;
;                                                                          ;
;       Registers used:   INCH                                            ;
;                            A,F,Y                                         ;
;                          INST                                            ;
;                            A,F                                           ;
;                          OUTCH                                           ;
;                            A,F,Y                                         ;
;                          OUTST                                           ;
;                            A,F                                           ;
;                          INIT                                            ;
;                            A,F                                           ;
;                                                                          ;
;       Time:              INCH                                            ;
;                            33 cycles if a character is available         ;
;                          INST                                            ;
;                            12 cycles                                     ;
;                          OUTCH                                           ;
;                            83 cycles if the output buffer is empty and   ;
;                               the VIA is ready to transmit               ;
;                          OUTST                                           ;
;                            12 cycles                                     ;
;                          INIT                                            ;
;                            93 cycles                                     ;
;                          IOSRVC                                          ;
;                            24 cycles minimum if the interrupt is not ours;
;                            43 cycles to service a input interrupt        ;
;                            81 cycles to service a output interrupt       ;
;                                                                          ;
;       Size:              Program 194 bytes                               ;
;                          Data      7 bytes                               ;
;                                                                          ;
;                                                                          ;

;EXAMPLE 6522 VIA PORT DEFINITIONS
VIA     .EQU    0C090H          ;VIA BASE ADDRESS
VIABDR  .EQU    VIA             ;VIA PORT B DATA REGISTER
VIAADR  .EQU    VIA+1           ;VIA PORT A DATA REGISTER, WITH HANDSHAKING
VIABDD  .EQU    VIA+2           ;VIA PORT B DATA DIRECTION REGISTER
VIAADD  .EQU    VIA+3           ;VIA PORT A DATA DIRECTION REGISTER
VIAACR  .EQU    VIA+11          ;VIA AUXILIARY CONTROL REGISTER
VIAPCR  .EQU    VIA+12          ;VIA PERIPHERAL CONTROL REGISTER
VIAIFR  .EQU    VIA+13          ;VIA INTERRUPT FLAG REGISTER
VIAIER  .EQU    VIA+14          ;VIA INTERRUPT ENABLE REGISTER

IRQVEC  .EQU    03FEH           ;APPLE IRQ VECTOR ADDRESS

;READ A CHARACTER
```

```
INCH:
        JSR     INST            ;GET INPUT STATUS
        BCC     INCH            ;WAIT IF CHARACTER IS NOT AVAILABLE
        PHP                     ;SAVE CURRENT STATE OF INTERRUPT SYSTEM
        SEI                     ;DISABLE INTERRUPTS
        LDA     RECDAT          ;GET THE CHARACTER FROM THE BUFFER
        LDA     #0
        STA     RECDF           ;INDICATE BUFFER IS NOW EMPTY
        LDA     RECDAT          ;GET THE CHARACTER FROM THE BUFFER
        PLP                     ;RESTORE FLAGS
        RTS

;RETURN INPUT STATUS (CARRY = 1 IF DATA IS AVAILABLE)
INST:
        LDA     RECDF           ;GET THE DATA READY FLAG
        LSR     A               ;SET CARRY FROM FLAG
                                ; CARRY = 1 IF CHARACTER IS AVAILABLE
        RTS

;WRITE A CHARACTER
OUTCH:
        PHP                     ;SAVE STATE OF INTERRUPT FLAG
        PHA                     ;SAVE CHARACTER TO OUTPUT

        ;WAIT FOR THE CHARACTER BUFFER TO EMPTY, THEN STORE THE NEXT CHARACTER
WAITOC:
        JSR     OUTST           ;GET THE OUTPUT STATUS
        BCS     WAITOC          ;WAIT IF THE OUTPUT BUFFER IS FULL
        SEI                     ;DISABLE INTERRUPTS WHILE LOOKING AT THE
                                ; SOFTWARE FLAGS
        PLA                     ;GET THE CHARACTER
        STA     TRNDAT          ;STORE THE CHARACTER
        LDA     #0FFH           ;INDICATE CHARACTER AVAILABLE (BUFFER FULL)
        STA     TRNDF
        LDA     OIE             ;HAS THE OUTPUT DEVICE ALREADY REQUESTED
                                ; SERVICE?
        BNE     OUTCH1          ; NO, BRANCH AND WAIT FOR AN INTERRUPT
        JSR     OUTDAT          ; YES, SEND THE DATA TO THE PORT NOW
OUTCH1: PLP                     ;RESTORE FLAGS
        RTS

;OUTPUT STATUS (CARRY = 1 IF BUFFER IS FULL)
OUTST:
        LDA     TRNDF           ;CARRY = 1 IF CHARACTER IS IN THE BUFFER
        LSR     A
        RTS

;INITIALIZE
INIT:
        PHP                     ;SAVE CURRENT STATE OF FLAGS
        SEI                     ;DISABLE INTERRUPTS

        ;INITIALIZE THE SOFTWARE FLAGS
```

```
        LDA     .#0
        STA     RECDF           ;NO INPUT DATA AVAILABLE
        STA     TRNDF           ;OUTPUT BUFFER EMPTY
        LDA     #0FFH           ;OUTPUT DEVICE HAS NOT REQUESTED SERVICE
        STA     OIE

        ;SAVE THE CURRENT IRQ VECTOR IN NEXTSR
        LDA     IRQVEC
        STA     NEXTSR
        LDA     IRQVEC+1
        STA     NEXTSR+1

        ;SET THE IRQ VECTOR TO OUR INPUT SERVICE ROUTINE
        LDA     AIOS
        STA     IRQVEC
        LDA     AIOS+1
        STA     IRQVEC+1

        ;INITIALIZE THE 6522 VIA
        LDA     #00000000B
        STA     VIAADD          ;SET PORT A TO INPUT
        LDA     #11111111B
        STA     VIABDD          ;SET PORT B TO OUTPUT
        LDA     #10001010B
        STA     VIAPCR          ;SET PORT A TO
                                ; INTERRUPT ON A LOW TO HIGH OF CA1 (BIT 0 = 1)
                                ; OUTPUT A LOW PULSE ON CA2 (BITS 1..3 = 101)
                                ;SET PORT B TO
                                ; INTERRUPT ON A LOW TO HIGH OF CB1 (BIT 4 = 1)
                                ; HANDSHAKE OUTPUT MODE (BITS 5..7 = 001)
        LDA     #00000001B
        STA     VIAACR          ;SET AUXILIARY CONTROL TO ENABLE INPUT LATCHING
                                ; FOR PORT A
        LDA     #00010010B      ;SET INTERRUPT ENABLE REGISTER TO ALLOW
                                ; INTERRUPTS ON CA1 (BIT 1) AND CB1 (BIT 4)
                                ;
        STA     VIAIER
        PLP                     ;RESTORE CURRENT STATE OF THE FLAGS
        RTS

AIOS:   .WORD   IOSRVC          ;ADDRESS OF INPUT OUTPUT SERVICE ROUTINE

;INPUT OUTPUT INTERRUPT SERVICE ROUTINE
IOSRVC:
        PHA                     ;SAVE REGISTER A
        CLD                     ;BE SURE PROCESSOR IS IN BINARY MODE

        ;GET THE VIA STATUS: BIT 1 = 1 IF AN INPUT INTERRUPT
        ;BIT 4 = 1 IF AN OUTPUT INTERRUPT
        LDA     VIAIFR
        AND     #10B            ;TEST BIT 1
        BNE     IINT            ;GOTO INPUT INTERRUPT IF BIT 1 = 1
        LDA     VIAIFR
        AND     #1000B          ;TEST BIT 4
        BNE     OINT            ;GOTO OUTPUT INTERRUPT IF BIT 4 = 1
```

```
        ;THE INTERRUPT WAS NOT CAUSED BY THIS VIA
        PLA
        JMP     (NEXTSR)        ;GOTO THE NEXT SERVICE ROUTINE

;SERVICE INPUT INTERRUPTS
IINT:
        LDA     VIAADR          ;READ THE DATA
                                ; (WHICH PULSES CA2 FOR THE HANDSHAKE AND
                                ;  CLEARS THE INTERRUPT FLAG)
        STA     RECDAT          ;STORE DATA
        LDA     #0FFH
        STA     RECDF           ;INDICATE WE HAVE A CHARACTER IN RECDAT
        JMP     EXIT            ;EXIT IOSRVC

;SERVICE OUTPUT INTERRUPTS
;NOTE THAT WE CAN CLEAR A 6522 INTERRUPT BY READING THE DATA
; REGISTER. THUS WE CAN CLEAR AN OUTPUT INTERRUPT WITHOUT
; SERVICING IT OR DISABLING IT. HOWEVER, IF WE DO THIS, WE
; MUST HAVE A FLAG (OIE) THAT INDICATES THE OUTPUT INTERRUPT
; HAS OCCURRED BUT HAS NOT BEEN SERVICED. OUTCH CAN THEN USE
; THE OIE FLAG TO DETERMINE WHETHER TO SEND THE DATA IMMEDIATELY
; OR WAIT FOR AN OUTPUT INTERRUPT TO SEND IT.
OINT:
        LDA     TRNDF           ;GET DATA AVAILABLE FLAG
        BNE     NODATA          ;BRANCH IF THERE IS NO DATA TO SEND
        JSR     OUTDAT          ; ELSE OUTPUT THE DATA
        JMP     EXIT

NODATA:
        LDA     VIABDR          ;READ THE PORT B DATA REGISTER TO CLEAR THE
                                ; INTERRUPT.
        LDA     #0              ;INDICATE OUTPUT INTERRUPT HAS OCCURRED
        STA     OIE             ; BUT HAS NOT BEEN SERVICED

EXIT:
        PLA                     ;RESTORE REGISTER A
        RTI                     ;RETURN FROM INTERRUPT

;*****************************************
;ROUTINE: OUTDAT
;PURPOSE: SEND A CHARACTER TO THE VIA
;ENTRY: TRNDAT = CHARACTER TO SEND
;EXIT:  NONE
;REGISTERS USED: A,F
;*****************************************

OUTDAT:
        LDA     TRNDAT          ;GET THE CHARACTER
        STA     VIABDR          ;OUTPUT DATA TO PORT B
        LDA     #0
        STA     TRNDF           ;INDICATE BUFFER EMPTY
        LDA     #0FFH
        STA     OIE             ;INDICATE NO UNSERVICED OUTPUT INTERRUPT
        RTS
```

```
;DATA SECTION
RECDAT  .BLOCK  1               ;RECEIVE DATA
RECDF   .BLOCK  1               ;RECEIVE DATA FLAG (0 = NO DATA, FF = DATA)
TRNDAT  .BLOCK  1               ;TRANSMIT DATA
TRNDF   .BLOCK  1               ;TRANSMIT DATA FLAG (0 = BUFFER EMPTY
                                ;                   FF = BUFFER FULL)
OIE     .BLOCK  1               ;OUTPUT INTERRUPT FLAG
                                ; (0  = INTERRUPT OCCURRED WITHOUT SERVICE
                                ;  FF = INTERRUPT SERVICED)
NEXTSR  .BLOCK  2               ;ADDRESS OF THE NEXT INTERRUPT SERVICE ROUTINE


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC1102:
        JSR     INIT            ;INITIALIZE
        CLI                     ;ENABLE INTERRUPTS

        ;SIMPLE EXAMPLE
LOOP:
        JSR     INCH            ;READ A CHARACTER
        PHA
        JSR     OUTCH           ;ECHO IT
        PLA
        CMP     #1BH            ;IS IT AN ESCAPE CHARACTER ?
        BNE     LOOP            ;STAY IN LOOP IF NOT
        BRK

        ;AN ASYNCHRONOUS EXAMPLE
        ; OUTPUT "A" TO THE CONSOLE CONTINUOUSLY BUT ALSO LOOK AT THE
        ; INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS.
ASYNLP:
        ;OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        JSR     OUTST           ;IS OUTPUT BUSY ?
        BCS     ASYNLP          ;BRANCH IF IT IS
        LDA     #"A"
        JSR     OUTCH           ;OUTPUT THE CHARACTER

        ;GET A CHARACTER FROM THE INPUT PORT IF ANY
        JSR     INST            ;IS INPUT DATA AVAILABLE ?
        BCC     ASYNLP          ;BRANCH IF NOT (SEND ANOTHER "A")
        JSR     INCH            ;GET THE CHARACTER
        CMP     #1BH            ;IS IT AN ESCAPE CHARACTER ?
        BEQ     DONE            ;BRANCH IF IT IS
        JSR     OUTCH           ;ELSE ECHO IT
        JMP     ASYNLP          ;AND CONTINUE

DONE:
        BRK
        JMP     SC1102

        .END    ;PROGRAM
```

Buffered Interrupt-Driven Input/Output Using a 6850 ACIA (SINTB)

Performs interrupt-driven input and output using a 6850 ACIA and multiple-character buffers. Consists of the following subroutines:

1. INCH reads a character from the input buffer.

2. INST determines whether there are any characters in the input buffer.

3. OUTCH writes a character into the output buffer.

4. OUTST determines whether the output buffer is full.

5. INIT initializes the buffers and the 6850 device.

6. IOSRVC determines which interrupt occurred and provides the proper input or output service.

Procedures:

1. INCH waits for a character to become available, gets the character from the head of the input buffer, moves the head of the buffer up one position, and decreases the input buffer counter by 1.

2. INST sets the Carry to 0 if the input buffer counter is zero and to 1 if the counter is non-zero.

3. OUTCH waits until there is empty space in the output buffer (that is, until the output buffer is not full), stores the character at the tail of the output buffer, moves the tail of the buffer up one position, and increases the output buffer counter by 1.

4. OUTST sets the Carry flag to 1 if the output buffer counter is equal to the buffer's length and to 0 if it is not.

Registers Used:
1. INCH: A, F, Y
2. INST: A, F
3. OUTCHA, F, Y
4. OUTST:A, F
5. INIT: A, F

Execution Time:
1. INCH: 70 cycles if a character is available
2. INST: 18 cycles
3. OUTCH: 75 cycles minimum, 105 cycles maximum if the output buffer is not full and the ACIA is ready to transmit
4. OUTST: 12 cycles
5. INIT: 89 cycles
6. IOSRVC: 73 cycles to service an input interrupt, 102 cycles to service an output interrupt, 27 cycles to determine the interrupt is from another device.

Program Size: 258 bytes

Data Memory Required: Seven bytes anywhere in RAM plus the input and output buffers. The seven bytes anywhere in RAM hold the input buffer counter (one byte at address ICNT), the index to the head of the input buffer (one byte at address IHEAD), the index to the tail of the input buffer (one byte at address ITAIL), the output buffer counter (one byte at address OCNT), the index to the head of the output buffer (one byte at address OHEAD), the index to the tail of the output buffer (one byte at address OIE), and an Output Interrupt Enable flag (one byte at address OIE). The input buffer starts at address IBUF and its size is IBSZ; the output buffer starts at address OBUF and its size is OBSZ.

5. INIT clears the buffer counters, sets both the heads and the tails of the buffers to zero, sets up the interrupt vector, resets the ACIA by performing a master reset on its control register (the ACIA has no reset input), and places the ACIA in its required operating mode by storing the appropriate

value in its control register. INIT enables the input interrupt and disables the output interrupt. It does, however, clear the output interrupt enable flag, thus indicating that the ACIA is ready to transmit data, although it cannot cause an output interrupt.

6. IOSRVC determines whether the interrupt was an input interrupt (bit 0 of the ACIA status register = 1), an output interrupt (bit 1 of the ACIA status register = 1), or the product of some other device. If the input interrupt occurred, the program reads the data and determines if there is room for it in the buffer. If there is room, the processor stores the character at the tail of the input buffer, moves the tail of the buffer up one position, and increases the input buffer counter by 1. If the output interrupt occurred, the program determines whether there is any data in the output buffer. If there is none, the program disables the output interrupt (so it will not interrupt repeatedly) and clears an Output Interrupt flag that indicates the ACIA is actually ready. The flag lets the main program know that the ACIA is ready even through it cannot declare its readiness by forcing an interrupt. If there is data in the output buffer, the program obtains a character from the head of the buffer, sends it to the ACIA, moves the head of the buffer up one position, and decreases the output buffer counter by 1. It then enables both input and output interrupts and sets the Output Interrupt flag (in case that flag had been cleared earlier).

The new problem that occurs in using multiple-character buffers is the management of queues. The main program must read the data in the same order in which the input interrupt service routine receives it. Similarly, the output interrupt service routine must send the data in the same order that the main program stores it. Thus we have the following requirements for handling input:

1. The main program must know whether there is anything in the input buffer.

2. If the input buffer is not empty, the main program must know where the oldest character is (that is, the one that was received first).

3. The input interrupt service routine must know whether the input buffer is full.

4. If the input buffer is not full, the input interrupt service routine must know where the next empty place is (that is, it must know where it should store the new character).

The output interrupt service routine and the main program have a similar set of requirements for the output buffer, although the roles of sender and receiver are reversed.

We meet requirements 1 and 3 by maintaining a counter ICNT. INIT initializes ICNT to zero, the interrupt service routine adds 1 to it whenever it receives a character (assuming the buffer is not full), and the main program subtracts 1 from it whenever it removes a character from the buffer (assuming the buffer is not empty). Thus the main program can determine whether the input buffer is empty by checking if ICNT is zero. Similarly, the interrupt service routine can determine whether the input buffer is full by checking if ICNT is equal to the size of the buffer.

We meet requirements 2 and 4 by maintaining two indexes, IHEAD and ITAIL, defined as follows:

1. ITAIL is the index of the next empty location in the buffer.

2. IHEAD is the index of the oldest character in the buffer.

INIT initializes IHEAD and ITAIL to zero. Whenever the interrupt service routine receives a character, it places it in the buffer at index ITAIL and increments ITAIL by 1 (assuming that the buffer is not full). Whenever the main program reads a character, it removes it from the buffer at index IHEAD and increments IHEAD by 1 (assuming that the buffer is not empty). Thus IHEAD "chases" ITAIL across the buffer with the service routine entering characters at one end (the tail) while the main program removes them from the other end (the head). The occupied part of the buffer thus could start and end anywhere. If either IHEAD or ITAIL reaches the physical end of the buffer, we simply set it back to zero. Thus we allow wraparound on the buffer; that is, the occupied part of the buffer could start near the end (say, at byte #195 of a 200-byte buffer) and continue back to the beginning (say, to byte #10). Thus IHEAD would be 195, ITAIL would be 10, and the buffer would contain 15 characters occupying bytes #195 through 199 and 0 through 9.

Entry Conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in accumulator
4. OUTST: none
5. INIT: none

Exit Conditions

1. INCH: character in accumulator
2. INST: Carry flag = 0 if no characters are available, 1 if a character is available
3. OUTCH: none
4. OUTST: Carry flag = 0 if output buffer is not full, 1 if it is full
5. INIT: none

```
;       Title           Interrupt input and output using a 6850     ;
;                       ACIA and a multiple character buffer.        ;
;       Name:           SINTB                                        ;
;                                                                    ;
;                                                                    ;
                                                                     ;
;                                                                    ;
;       Purpose:        This program consists of 5 subroutines which ;
;                       perform interrupt driven input and output using ;
;                       a 6850 ACIA.                                 ;
;                                                                    ;
;                       INCH                                         ;
;                         Read a character.                          ;
;
```

```
;                         INST                                        ;
;                           Determine input status (whether a character ;
;                           is available).                            ;
;                         OUTCH                                       ;
;                           Write a character.                        ;
;                         OUTST                                       ;
;                           Determine output status (whether the output ;
;                           buffer is full).                          ;
;                         INIT                                        ;
;                           Initialize.                               ;
;                                                                     ;
;       Entry:            INCH                                        ;
;                           No parameters.                            ;
;                         INST                                        ;
;                           No parameters.                            ;
;                         OUTCH                                       ;
;                           Register A = character to transmit        ;
;                         OUTST                                       ;
;                           No parameters.                            ;
;                         INIT                                        ;
;                           No parameters.                            ;
;                                                                     ;
;       Exit:             INCH                                        ;
;                           Register A = character.                   ;
;                         INST                                        ;
;                           Carry flag equals 0 if no characters are  ;
;                           available, 1 if character is available.   ;
;                         OUTCH                                       ;
;                           No parameters                             ;
;                         OUTST                                       ;
;                           Carry flag equals 0 if output buffer is   ;
;                           empty, 1 if it is full.                   ;
;                         INIT                                        ;
;                           No parameters.                            ;
;                                                                     ;
;       Registers used: INCH                                          ;
;                           A,F,Y                                     ;
;                         INST                                        ;
;                           A,F                                       ;
;                         OUTCH                                       ;
;                           A,F,Y                                     ;
;                         OUTST                                       ;
;                           A,F                                       ;
;                         INIT                                        ;
;                           A,F                                       ;
;                                                                     ;
;                                                                     ;
;       Registers used: INCH                                          ;
;                           A,F,Y                                     ;
;                         INST                                        ;
;                           A,F                                       ;
;                         OUTCH                                       ;
;                           A,F,Y                                     ;
;                         OUTST                                       ;
;                           A,F                                       ;
;                         INIT                                        ;
```

```
;                       A,F                                             ;
;                                                                       ;
;       Time:           INCH                                            ;
;                         70 cycles if a character is available         ;
;                       INST                                            ;
;                         18 cycles                                     ;
;                       OUTCH                                           ;
;                         75 cycles minimum, if the output buffer is    ;
;                            not full and the ACIA is ready to transmit ;
;                       OUTST                                           ;
;                         12 cycles                                     ;
;                       INIT                                            ;
;                         89 cycles                                     ;
;                       IOSRVC                                          ;
;                         27 cycles minimum if the interrupt is not ours;
;                         73 cycles to service a input interrupt        ;
;                         102 cycles to service a output interrupt      ;
;                                                                       ;
;       Size:           Program 258 bytes                               ;
;                       Data      7 bytes plus size of buffers          ;
;                                                                       ;
;       Buffers:                                                        ;
;                       The routines assume two buffers starting at     ;
;                       addresses IBUF and OBUF. The lengths of the     ;
;                       buffers in bytes are IBSZ and OBSZ. For the     ;
;                       input buffer, IHEAD is the index of the oldest  ;
;                       character (the next one the main program should ;
;                       read), ITAIL is the index of the next empty     ;
;                       element (the next one the service routine       ;
;                       should fill), and ICNT is the number of bytes   ;
;                       currently filled with characters. For the       ;
;                       output buffer, OHEAD is the index of the oldest ;
;                       character (the next one the service routine     ;
;                       should send), OTAIL is the index of the next    ;
;                       empty element (the next one the main program    ;
;                       should fill), and OCNT is the number of bytes   ;
;                       currently filled with characters.               ;
;                                                                       ;
;       Note:           Wraparound is provided on both buffers, so that ;
;                       the currently filled area may start anywhere    ;
;                       and extend through the end of the buffer and    ;
;                       back to the beginning. For example, if the      ;
;                       output buffer is 40 hex bytes long, the section ;
;                       filled with characters could exetend from       ;
;                       OBUF+32H (OHEAD=32H) to OBUF+10H (OTAIL=11H).   ;
;                       That is, there are 19H filled bytes occupying   ;
;                       addresses OBUF+32H through OBUF+39H and         ;
;                       continuing to OBUF through OBUF+10H. The buffer ;
;                       thus looks like a television picture with the   ;
;                       vertical hold skewed, so that the frame starts  ;
;                       above the bottom of the screen, leaves off at   ;
;                       the top, and continues at the bottom.           ;
;                                                                       ;
;                                                                       ;

;EXAMPLE 6850 ACIA PORT DEFINITIONS FOR AN APPLE SERIAL BOARD IN SLOT 1
```

```
ACIASR  .EQU    0C094H          ;ACIA STATUS REGISTER
ACIADR  .EQU    0C095H          ;ACIA DATA REGISTER
ACIACR  .EQU    0C094H          ;ACIA CONTROL REGISTER
IRQVEC  .EQU    03FEH           ;APPLE IRQ VECTOR ADDRESS

;READ A CHARACTER
INCH:
        JSR     INST            ;IS A CHARACTER AVAILABLE ?
        BCC     INCH            ;BRANCH IF NOT
        PHP                     ;SAVE CURRENT STATE OF INTERRUPTS
        SEI                     ;DISABLE INTERRUPTS
        LDY     IHEAD
        LDA     IBUF,Y          ;GET CHARACTER AT HEAD OF BUFFER
        INY
        CPY     #IBSZ           ;DO WE NEED WRAPAROUND IN BUFFER ?
        BCC     INCH1           ;BRANCH IF NOT
        LDY     #0              ;ELSE SET HEAD BACK TO ZERO
INCH1:
        STY     IHEAD
        DEC     ICNT            ;DECREMENT CHARACTER COUNT
        PLP                     ;RESTORE FLAGS
        RTS

;RETURN INPUT STATUS (CARRY = 1 IF CHARACTERS ARE AVAILABLE, 0 IF NOT)
INST:
        CLC                     ;CLEAR CARRY (ASSUME NO CHARACTERS AVAILABLE)
        LDA     ICNT
        BEQ     INST1           ;BRANCH IF THERE ARE NONE
        SEC                     ;CARRY = 1 (CHARACTERS ARE AVAILABLE)
INST1:
        RTS

;WRITE A CHARACTER
OUTCH:
        PHP                     ;SAVE STATE OF INTERRUPT FLAG
        PHA                     ;SAVE CHARACTER TO OUTPUT

        ;WAIT UNTIL THERE IS EMPTY SPACE IN THE OUTPUT BUFFER
WAITOC:
        JSR     OUTST           ;IS THE OUTPUT BUFFER FULL ?
        BCS     WAITOC          ;BRANCH IF IT IS FULL
        SEI                     ;DISABLE INTERRUPTS WHILE LOOKING AT THE
                                ; SOFTWARE FLAGS
        PLA                     ;GET THE CHARACTER
        LDY     OTAIL
        STA     OBUF,Y          ;STORE CHARACTER IN THE BUFFER
        INY
        CPY     #OBSZ           ;DO WE NEED WRAPAROUND ON THE BUFFER ?
        BCC     OUTCH1          ;BRANCH IF NOT
        LDY     #0              ;ELSE SET TAIL BACK TO ZERO
OUTCH1:
        STY     OTAIL
        INC     OCNT            ;INCREMENT BUFFER COUNTER
        LDA     OIE             ;ARE INTERRUPTS DISABLED BUT THE ACIA IS
                                ; ACTUALLY READY ?
        BNE     OUTCH2          ;EXIT IF ACIA INTERRUPTS NOT READY AND ENABLED
```

```
        JSR     OUTDAT              ;ELSE SEND THE DATA TO THE PORT AND ENABLE
                                    ; INTERRUPTS
OUTCH2:
        PLP                         ;RESTORE FLAGS
        RTS


;OUTPUT STATUS
OUTST:
        LDA     OCNT
        CMP     #OBSZ               ;IS OUTPUT BUFFER FULL ?
                                    ; IF OCNT >= OBSZ THEN
                                    ;    CARRY = 1 INDICATING THAT THE OUTPUT
                                    ;              BUFFER IS FULL
                                    ; ELSE
                                    ;    CARRY = 0 INDICATING THAT THE CHARACTER
                                    ;              CAN BE PLACED IN THE BUFFER
        RTS


;INITIALIZE
INIT:
        PHP                         ;SAVE CURRENT STATE OF FLAGS
        SEI                         ;DISABLE INTERRUPTS

        ;INITIALIZE THE SOFTWARE FLAGS
        LDA     #0
        STA     ICNT                ;NO INPUT DATA
        STA     IHEAD
        STA     ITAIL
        STA     OCNT                ;NO OUTPUT DATA
        STA     OHEAD
        STA     OTAIL
        STA     OIE                 ;ACIA IS READY TO TRANSMIT (NOTE THIS !!)

        ;SAVE THE CURRENT IRQ VECTOR IN NEXTSR
        LDA     IRQVEC
        STA     NEXTSR
        LDA     IRQVEC+1
        STA     NEXTSR+1

        ;SET THE IRQ VECTOR TO OUR INPUT SERVICE ROUTINE
        LDA     AIOS
        STA     IRQVEC
        LDA     AIOS+1
        STA     IRQVEC+1

        ;INITIALIZE THE 6850 ACIA
        LDA     #011B
        STA     ACIACR              ;MASTER RESET ACIA
        LDA     #10010001B
        STA     ACIACR              ;INITIALIZE ACIA MODE TO
                                    ;  DIVIDE BY 16
                                    ;  8 DATA BITS
                                    ;  2 STOP BITS
```

```
                                ;  OUTPUT INTERRUPTS DISABLED (NOTE THIS !!)
                                ;  INPUT INTERRUPTS ENABLED

        PLP                     ;RESTORE CURRENT STATE OF THE FLAGS
        RTS

AIOS:   .WORD   IOSRVC          ;ADDRESS OF INPUT OUTPUT SERVICE ROUTINE

;INPUT OUTPUT INTERRUPT SERVICE ROUTINE
IOSRVC:
        PHA                     ;SAVE REGISTER A
        CLD                     ;BE SURE PROCESSOR IS IN BINARY MODE

        ;GET THE ACIA STATUS: BIT 0 = 1 IF AN INPUT INTERRUPT
        ;BIT 1 = 1 IF AN OUTPUT INTERRUPT
        LDA     ACIASR
        LSR     A               ;BIT 0 TO CARRY
        BCS     IINT            ;BRANCH IF AN INPUT INTERRUPT
        LSR     A               ;BIT 1 TO CARRY
        BCS     OINT            ;BRANCH IF AN OUTPUT INTERRUPT

        ;THE INTERRUPT WAS NOT OURS
        PLA
        JMP     (NEXTSR)        ;GOTO THE NEXT SERVICE ROUTINE

;SERVICE INPUT INTERRUPTS
IINT:
        TYA
        PHA                     ;SAVE REGISTER Y

        ;GET THE DATA AND STORE IT IN THE BUFFER IF THERE IS ROOM
        LDA     ACIADR          ;READ THE DATA
        LDY     ICNT            ;IS THERE ROOM IN THE BUFFER ?
        CPY     #IBSZ
        BCS     EXIT            ;EXIT, NO ROOM IN THE BUFFER
        LDY     ITAIL           ;ELSE STORE THE DATA IN THE BUFFER
        STA     IBUF,Y
        INY                     ;INCREMENT TAIL INDEX
        CPY     #IBSZ           ;DO WE NEED WRAPAROUND ON THE BUFFER ?
        BCC     IINT1           ;BRANCH IF NOT
        LDY     #0              ;ELSE SET TAIL BACK TO ZERO
IINT1:
        STY     ITAIL           ;STORE NEW TAIL INDEX
        INC     ICNT            ;INCREMENT INPUT BUFFER COUNTER
        JMP     EXIT            ;EXIT IOSRVC

;SERVICE OUTPUT INTERRUPTS
OINT:
        TYA
        PHA                     ;SAVE REGISTER Y

        LDA     OCNT            ;IS THERE ANY DATA IN THE OUTPUT BUFFER ?
        BEQ     NODATA          ;BRANCH IF NOT (DISABLE THE INTERRUPTS)
        JSR     OUTDAT          ;ELSE SEND A CHARACTER
        JMP     EXIT
```

```
NODATA:
        LDA     #10010001B      ;DISABLE OUTPUT INTERRUPTS, ENABLE INPUT
                                ; INTERRUPTS, 8 DATA BITS, 2 STOP BITS, DIVIDE
                                ; BY 16 CLOCK
        STA     ACIACR          ;TURN OFF INTERRUPTS
        LDA     #0
        STA     OIE             ;INDICATE OUTPUT INTERRUPTS ARE DISABLED
                                ; BUT ACIA IS ACTUALLY READY
EXIT:
        PLA
        TAY                     ;RESTORE REGISTER Y
        PLA                     ;RESTORE REGISTER A
        RTI                     ;RETURN FROM INTERRUPT

;****************************************
;ROUTINE: OUTDAT
;PURPOSE: SEND A CHARACTER TO THE ACIA FROM THE OUTPUT BUFFER
;ENTRY: OHEAD IS THE INDEX INTO OBUF OF THE CHARACTER TO SEND
;EXIT:  NONE
;REGISTERS USED: A,F
;****************************************

OUTDAT:
        LDA     ACIASR
        AND     #00000010B      ;IS ACIA OUTPUT REGISTER EMPTY ?
        BEQ     OUTDAT          ;BRANCH IF NOT EMPTY (BIT 1 = 0)
        LDY     OHEAD
        LDA     OBUF,Y          ;GET THE CHARACTER FROM THE BUFFER
        STA     ACIADR          ;SEND THE DATA
        INY
        CPY     #OBSZ           ;DO WE NEED WRAPAROUND ON THE BUFFER ?
        BCC     OUTD1           ;BRANCH IF NOT
        LDY     #0              ;ELSE SET HEAD BACK TO ZERO
OUTD1:
        STY     OHEAD           ;SAVE NEW HEAD INDEX
        DEC     OCNT            ;DECREMENT OUTPUT BUFFER COUNTER
        LDA     #10110001B
        STA     ACIACR          ;ENABLE 6850 OUTPUT AND INPUT INTERRUPTS,
                                ; 8 DATA BITS, 2 STOP BITS, DIVIDE BY 16 CLOCK
        LDA     #0FFH
        STA     OIE             ;INDICATE THE OUTPUT INTERRUPTS ARE ENABLED

        RTS


;DATA SECTION
ICNT    .BLOCK  1               ;INPUT BUFFER COUNTER
IHEAD   .BLOCK  1               ;INDEX TO HEAD OF INPUT BUFFER
ITAIL   .BLOCK  1               ;INDEX TO TAIL OF INPUT BUFFER
OCNT    .BLOCK  1               ;OUTPUT BUFFER COUNTER
OHEAD   .BLOCK  1               ;INDEX TO HEAD OF OUTPUT BUFFER
OTAIL   .BLOCK  1               ;INDEX TO TAIL OF OUTPUT BUFFER
OIE     .BLOCK  1               ;OUTPUT INTERRUPT ENABLE FLAG

IBSZ    .EQU    80              ;INPUT BUFFER SIZE
IBUF    .BLOCK  IBSZ            ;INPUT BUFFER
```

```
OBSZ    .EQU    80                  ;OUTPUT BUFFER SIZE
OBUF    .BLOCK  OBSZ                ;OUTPUT BUFFER
NEXTSR  .BLOCK  2                   ;ADDRESS OF THE NEXT INTERRUPT SERVICE ROUTINE


;                                                                   ;
;                                                                   ;
;       SAMPLE EXECUTION:                                           ;
;                                                                   ;
;                                                                   ;


SC1103:
        JSR     INIT                ;INITIALIZE
        CLI                         ;ENABLE INTERRUPTS

        ;SIMPLE EXAMPLE
LOOP:
        JSR     INCH                ;READ A CHARACTER
        PHA
        JSR     OUTCH               ;ECHO IT
        PLA
        CMP     #1BH                ;IS CHARACTER AN ESCAPE ?
        BNE     LOOP                ;BRANCH IF NOT, CONTINUE LOOPING
        BRK

        ;AN ASYNCHRONOUS EXAMPLE
        ; OUTPUT "A" TO THE CONSOLE CONTINUOUSLY BUT ALSO LOOK AT THE
        ; INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS.
ASYNLP:
        ;OUTPUT AN "A" IF OUTPUT IS NOT BUSY

        JSR     OUTST               ;IS OUTPUT BUSY ?
        BCS     ASYNLP              ;BRANCH IF IT IS
        LDA     #"A"
        JSR     OUTCH               ;OUTPUT THE CHARACTER

        ;GET A CHARACTER FROM THE INPUT PORT IF ANY
        JSR     INST                ;IS INPUT AVAILABLE ?
        BCC     ASYNLP              ;BRANCH IF NOT (SEND ANOTHER "A")
        JSR     INCH                ;GET THE CHARACTER
        CMP     #1BH                ;IS CHARACTER AN ESCAPE ?
        BEQ     DONE                ;BRANCH IF IT IS
        JSR     OUTCH               ;ELSE ECHO IT
        JMP     ASYNLP              ; AND CONTINUE

DONE:
        BRK

        .END    ;PROGRAM
```

Maintains a time-of-day 24-hour clock and a calendar based on a real-time clock interrupt. Consists of the following subroutines:

1. CLOCK returns the starting address of the clock variables.

2. ICLK initializes the clock interrupt and initializes the clock variables to their default values.

3. CLKINT updates the clock after each interrupt (assumed to be spaced one tick apart).

A long example in the listing describes a time display routine for the Apple II computer. The routine prompts the operator for an initial date and time. It then continuously displays the date and time in the center of the monitor screen. The routine assumes an interrupt board in slot 2.

Procedure:

1. CLOCK loads the starting address of the clock variables into the accumulator (more significant byte) and index register Y (less significant byte). The clock variables are stored in the following order (lowest address first): ticks, seconds, minutes, hours, days, months, less significant byte of year, more significant byte of year.

2. ICLK loads the clock variables with their default values (8 bytes starting at address DFLTS) and initializes the clock interrupt (this would be mostly system-dependent).

3. CLKINT decrements the remaining tick count by one and updates the rest of the clock if necessary. Of course, the number of seconds and minutes must be less than 60 and the number of hours must be less than 24. The day of the month must be less than or equal to the last day for the current month; an array of the last days of each month begins at address LASTDY. If the month is February (that is, month 2), the program must check to see if the current year is a leap year. This requires a determination of whether the two least significant bits of memory location YEAR are both zeros. If the current year is a leap year, the last day of February is the 29th, not the 28th. The month number may not exceed 12 (December) or a carry to the year number is necessary. The program must reinitialize the variables properly when carries occur; that is, TICK to DTICK; seconds, minutes, and hours to zero; day and month to 1 (meaning the first day and January, respectively).

Registers Used:

1. CLOCK: A, F, Y
2. ICLK: A, Y
3. CLKINT: none

Execution Time:

1. CLOCK: 14 cycles
2. ICLK: 166 cycles
3. CLKINT: 33 cycles if only TICK must be decremented, 184 maximum if changing to a new year.

Program Size:

1. CLOCK: 7 bytes
2. ICLK: 39 bytes
3. CLKINT: 145 bytes

Data Memory Required: 18 bytes anywhere in RAM. These include eight bytes for the clock variables (starting at address ACVAR), eight bytes for the defaults (starting at address DFLTS), and two bytes for the address of the next service routine (starting at address NEXTSR).

Entry Conditions

1. CLOCK: none
2. ICLK: none
3. CLKINT: none

Exit Conditions

1. CLOCK: more significant byte of starting address of clock variables in accumulator, less significant byte in register Y
2. ICLK: none
3. CLKINT: none

Examples

These examples assume that the tick rate is DTICK Hz (less than 256 Hz — typical values would be 60 Hz or 100 Hz) and that the clock and calendar are saved in memory locations

TICK	number of ticks remaining before a carry occurs, counted down from DTICK
SEC	seconds (0 to 59)
MIN	minutes (0 to 59)
HOUR	hour of day (0 to 23)
DAY	day of month (1 to 28, 30, or 31, depending on month)
MONTH	month of year (1 through 12 for January through December)
YEAR & YEAR+1	current year

1. Starting values are March 7, 1982. 11:59.59 and 1 tick left.

That is,

(TICK) = 1
(SEC) = 59
(MIN) = 59
(HOUR) = 23
(DAY) = 07
(MONTH) = 03
(YEAR) = 1982

Result (after the tick): March 8, 1982 12:00.00 and DTICK ticks

That is,

(TICK) = DTICK
(SEC) = 0
(MIN) = 0
(HOUR) = 0
(DAY) = 08
(MONTH) = 03
(YEAR) = 1982

2. Starting values are Dec. 31, 1982. 11:59.59 p.m. and 1 tick left

That is,

(TICK) = 1
(SEC) = 59
(MIN) = 59
(HOUR) = 23
(DAY) = 31
(MONTH) = 12
(YEAR) = 1982

Result (after the tick): Jan. 1, 1983. 12:00.00 a.m. and DTICK ticks

That is,

(TICK) = DTICK
(SEC) = 0
(MIN) = 0
(HOUR) = 0
(DAY) = 1
(MONTH) = 1
(YEAR) = 1983

```
;       Title               Real time clock and calendar                 ;
;       Name:               CLOCK                                        ;
;                                                                        ;
;                                                                        ;
                                                                         ;
;                                                                        ;
;       Purpose:            This program maintains a time of day 24 hour ;
;                           clock and a calendar based on a real time clock ;
;                           interrupt.                                   ;
;                                                                        ;
;                           CLOCK                                        ;
;                             Returns the address of the clock variables ;
;                           ICLK                                         ;
;                             Initialize the clock interrupt             ;
;                                                                        ;
;       Entry:              CLOCK                                        ;
;                             None                                       ;
;                           ICLK                                         ;
;                             None                                       ;
;                                                                        ;
;       Exit:               CLOCK                                        ;
;                             Register A = High byte of the address of the ;
;                                          time variables.               ;
;                             Register Y = Low byte of the address of the ;
;                                          time variables.               ;
;                           ICLK                                         ;
;                             None                                       ;
;                                                                        ;
;       Registers used: All                                              ;
;                                                                        ;
;       Time:               CLOCK                                        ;
;                             14 cycles                                  ;
;                           ICLCK                                        ;
;                             166 cycles                                 ;
;                           CLKINT                                       ;
;                             22 cycles minimum if the interrupt is not ours;
;                             33 cycles normally if decrementing tick    ;
;                            184 cycles maximum if changing to a new year ;
;                                                                        ;
;       Size:               Program 191 bytes                            ;
;                           Data     18 bytes                            ;
;                                                                        ;
;                                                                        ;


IRQVEC: .EQU    03FEH           ;APPLE IRQ VECTOR
CLKPRT: .EQU    0C0A0H          ;SLOT 2 IO LOCATION OF AN INTERRUPT BOARD
CLKIM:  .EQU    01H             ;BIT 0 = INTERRUPT REQUEST BIT
TRUE:   .EQU    0FFH            ;NOT ZERO = TRUE
FALSE:  .EQU    0               ;ZERO = FALSE

;RETURN ADDRESS OF THE CLOCK VARIABLES
CLOCK:
        LDA     ACVAR+1
        LDY     ACVAR           ;GET ADDRESS OF CLOCK VARIABLES
        RTS
```

```
;INITIALIZE CLOCK INTERRUPT
ICLK:
        PHP                         ;SAVE FLAGS
        SEI                         ;DISABLE INTERRUPTS

        ;INITIALIZE CLOCK VARIABLES TO THE DEFAULT VALUEs
        LDY     #8
ICLK1:
        LDA     DFLTS-1,Y
        STA     CLKVAR-1,Y
        DEY
        BNE     ICLK1

        ;SAVE CURRENT IRQ VECTOR
        LDA     IRQVEC
        STA     NEXTSR
        LDA     IRQVEC+1
        STA     NEXTSR

        ;SET IRQ VECTOR TO CLKINT
        LDA     ACINT
        STA     IRQVEC
        LDA     ACINT+1
        STA     IRQVEC+1

        ;HERE SHOULD BE CODE TO INITIALIZE INTERRUPT HARDWARE

        ;EXIT
        PLP                         ;RESTORE FLAGS
        RTS

;HANDLE THE CLOCK INTERRUPT
CLKINT:
        PHA                         ;SAVE REGISTER A
        CLD                         ;BE SURE PROCESSOR IS IN BINARY MODE

        ;CHECK IF THIS IS OUR INTERRUPT
        ;  THIS IS AN EXAMPLE ONLY
        LDA     CLKPRT
        AND     #CLKIM              ;LOOK AT THE INTERRUPT REQUEST BIT
        BNE     OURINT              ;BRANCH IF IS OUR INTERRUPT
        PLA                         ;RESTORE REGISTER A
        JMP     (NEXTSR)            ;WAS NOT OUR INTERRUPT,
                                    ; TRY NEXT SERVICE ROUTINE

        ;PROCESS OUR INTERRUPT
OURINT:
        DEC     TICK
        BNE     EXIT1               ;BRANCH IF TICK DOES NOT EQUAL ZERO YET
                                    ; EXIT1 RESTORES ONLY REGISTER A
        LDA     DTICK
        STA     TICK                ;RESET TICK TO DEFAULT VALUE

        ;SAVE X AND Y NOW ALSO
        TYA
        PHA
```

```
        TXA
        PHA

        ;INCREMENT SECONDS
        INC     SEC
        LDA     SEC
        CMP     #60             ;SECONDS = 60 ?
        BCC     EXIT            ;EXIT IF LESS THAN 60 SECONDS
        LDY     #0              ;ELSE
        STY     SEC             ; ZERO SECONDS, GO TO NEXT MINUTE

        ;INCREMENT MINUTES
        INC     MIN
        LDA     MIN
        CMP     #60             ;MINUTES = 60 ?
        BCC     EXIT            ;EXIT IF LESS THAN 60 MINUTES
        STY     MIN             ;ELSE
                                ; ZERO MINUTES, GO TO NEXT HOUR

        ;INCREMENT HOURS
        INC     HOUR
        LDA     HOUR
        CMP     #24             ;HOURS = 24 ?
        BCC     EXIT            ;EXIT IF LESS THAN 24 HOURS
        STY     HOUR            ;ELSE
                                ; ZERO HOURS, GO TO NEXT DAY

        ;INCREMENT DAYS
        INC     DAY
        LDA     DAY
        LDX     MONTH           ;GET CURRENT MONTH
        CMP     LASTDY-1,X      ;DAY = LAST DAY OF THE MONTH ?
        BCC     EXIT            ;EXIT IF LESS THAN LAST DAY

        ;INCREMENT MONTH (HANDLE 29TH OF FEBRUARY)
        CPX     #2              ;IS THIS FEBRUARY ?
        BNE     INCMTH          ;BRANCH IF NOT FEBRUARY
        LDA     YEAR            ;IS IT A LEAP YEAR?
        AND     #00000011B
        BNE     INCMTH          ;BRANCH IF YEAR IS NOT LEAP YEAR

        ;THIS IS A FEBRUARY AND A LEAP YEAR SO 29 DAYS NOT 28 DAYS
        LDA     DAY
        CMP     #29
        BEQ     EXIT            ;EXIT IF NOT 29TH OF FEBRUARY

INCMTH:
        LDY     #1
        STY     DAY             ;CHANGE DAY TO 1, INCREMENT MONTH
        INC     MONTH
        LDA     MONTH
        CMP     #13             ;DONE WITH DECEMBER ?
        BCC     EXIT            ;EXIT IF NOT
        STY     MONTH           ;ELSE
                                ; CHANGE MONTH TO 1 (JANUARY)
```

```
        ;INCREMENT YEAR
        INC     YEAR                ;INCREMENT LOW BYTE
        BNE     EXIT
        INC     YEAR+1              ;INCREMENT HIGH BYTE
EXIT:
        ;RESTORE REGISTERS
        PLA
        TAX
        PLA
        TAY
EXIT1:
        PLA
        RTI                         ;RETURN FROM INTERRUPT

;ARRAY OF THE LAST DAYS OF EACH MONTH
LASTDY:
        .BYTE   31                  ;JANUARY
        .BYTE   28                  ;FEBRUARY (EXCEPT LEAP YEARS)
        .BYTE   31                  ;MARCH
        .BYTE   30                  ;APRIL
        .BYTE   31                  ;MAY
        .BYTE   30                  ;JUNE
        .BYTE   31                  ;JULY
        .BYTE   31                  ;AUGUST
        .BYTE   30                  ;SEPTEMBER
        .BYTE   31                  ;OCTOBER
        .BYTE   30                  ;NOVEMBER
        .BYTE   31                  ;DECEMBER

;CLOCK VARIABLES
ACVAR:  .WORD   CLKVAR              ;BASE ADDRESS OF CLOCK VARIABLES
CLKVAR:
TICK:   .BLOCK  1                   ;TICKS LEFT IN CURRENT SECOND
SEC:    .BLOCK  1                   ;SECONDS
MIN:    .BLOCK  1                   ;MINUTES
HOUR:   .BLOCK  1                   ;HOURS
DAY:    .BLOCK  1                   ;DAY = 1 THROUGH NUMBER OF DAYS IN A MONTH
MONTH:  .BLOCK  1                   ;MONTH 1=JANUARY .. 12=DECEMBER
YEAR:   .WORD   0                   ;YEAR

;DEFAULTS
DFLTS:
DTICK:  .BYTE   60                  ;DEFAULT TICK (60HZ INTERRUPT)
DSEC:   .BYTE   0                   ;DEFAULT SECONDS
DMIN:   .BYTE   0                   ;DEFAULT MINUTES
DHR:    .BYTE   0                   ;DEFAULT HOURS
DDAY:   .BYTE   1                   ;DEFAULT DAY
DMTH:   .BYTE   1                   ;DEFAULT MONTH
DYEAR:  .WORD   1981                ;DEFAULT YEAR

NEXTSR: .BLOCK  2                   ;ADDRESS OF THE NEXT INTERRUPT SERVICE ROUTINE
ACINT:  .WORD   CLKINT              ;ADDRESS OF THE CLOCK INTERRUPT ROUTINE
```

```
;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;     This routine prompts the operator for an initial date and time,   ;
;     it then continuously displays the date and time in the center of  ;
;     the screen.                                                       ;
;                                                                       ;
;     The operator may use the escape key to abort the routine. Any     ;
;     other key will reprompt for another initial date and time.        ;
;                                                                       ;
;                                                                       ;


;CLK VARIABLE OFFSETS
OTICK:  .EQU    0               ;OFFSET TO TICK
OSEC:   .EQU    1               ;OFFSET TO SECONDS
OMIN:   .EQU    2               ;OFFSET TO MINUTES
OHR:    .EQU    3               ;OFFSET TO HOURS
ODAY:   .EQU    4               ;OFFSET TO DAY
OMTH:   .EQU    5               ;OFFSET TO MONTH
OYEAR:  .EQU    6               ;OFFSET TO YEAR

;PAGE ZERO TEMPORARY
CVARS:  .EQU    0D0H            ;PAGE ZERO TEMPORARY FOR THE CLOCK VARIABLES
                                ; ADDRESS

;APPLE EQUATES FOR THE EXAMPLE
ESC:    .EQU    1BH             ;ESCAPE CHARACTER
CH      .EQU    24H             ;APPLE MONITOR CURSOR HORIZONTAL POSITION
CV      .EQU    25H             ;APPLE MONITOR CURSOR VERTICAL POSITION
HOME:   .EQU    0FC58H          ;APPLE MONITOR HOME ROUTINE
VTAB:   .EQU    0FC22H          ;APPLE MONITOR VTAB ROUTINE
RCHAR:  .EQU    0FD0CH          ;APPLE MONITOR CHARACTER INPUT ROUTINE
COUT:   .EQU    0FDEDH          ;APPLE MONITOR CHARACTER OUTPUT ROUTINE
GETLN1: .EQU    0FD6FH          ;APPLE MONITOR GET LINE WITH OUR PROMPT ROUTINE


SC1104:
        JSR     ICLK            ;INITIALIZE

        ;GET TODAYS DATE AND TIME MM/DD/YY HR:MIN:SEC
        ; PRINT PROMPT
PROMPT:
        JSR     HOME            ;HOME AND CLEAR SCREEN
        LDA     #0
        STA     MSGIDX
PMTLP:
        LDY     MSGIDX
        LDA     MSG,Y
        BEQ     RDTIME          ;BRANCH IF END OF MESSAGE
        INC     MSGIDX          ;INCREMENT TO NEXT CHARACTER
        JSR     WRCHAR          ;OUTPUT CHARACTER THROUGH APPLE MONITOR
        JMP     PMTLP           ;CONTINUE

        ;READ THE TIME STRING
```

```
RDTIME:
        JSR     RDLINE          ;READ A LINE INTO THE APPLE LINE BUFFER AT
                                ; 200H. RETURNS WITH LENGTH IN X

        ;GET THE ADDRESS OF THE CLOCK VARIABLES
        JSR     CLOCK           ;GET CLOCK VARIABLES
        STA     CVARS+1
        STY     CVARS           ;STORE ADDRESS

        ;INITIALIZE VARIABLES FOR READING NUMBERS
        STX     LLEN            ;SAVE LENGTH OF LINE
        LDA     #0
        STA     LIDX            ;INITIALIZE LINE INDEX TO ZERO

        ;GET MONTH
        JSR     NXTNUM          ;GET NEXT NUMBER FROM INPUT LINE
        LDY     #OMTH
        STA     (CVARS),Y       ;SET MONTH

        ;GET DAY
        JSR     NXTNUM
        LDY     #ODAY
        STA     (CVARS),Y

        ;GET YEAR
        JSR     NXTNUM
        LDY     #OYEAR
        STA     (CVARS),Y
        CLC
        ADC     CEN20           ;ADD 1900 TO ENTRY
        STA     (CVARS),Y       ;SET LOW BYTE OF YEAR
        LDA     CEN20+1
        ADC     #0
        INY
        STA     (CVARS),Y       ;SET HIGH BYTE OF YEAR

        ;GET HOUR
        JSR     NXTNUM
        LDY     #OHR
        STA     (CVARS),Y

        ;GET MINUTES
        JSR     NXTNUM
        LDY     #OMIN
        STA     (CVARS),Y

        ;GET SECONDS
        JSR     NXTNUM
        LDY     #OSEC
        STA     (CVARS),Y

        ;ENABLE INTERRUPTS
        CLI                     ;ENABLE INTERRUPTS

        ;HOME AND CLEAR THE SCREEN
```

```
        JSR     HOME

        ;LOOP PRINTING THE TIME EVERY SECOND
        ;MOVE CURSOR TO LINE 12 CHARACTER 12
LOOP:
        LDA     #11
        STA     CV              ;SET CURSOR VERTICAL POSITION
        STA     CH              ;SET CURSOR HORIZONTAL POSITION
        JSR     VTAB            ;POSITION CURSOR

        ;PRINT MONTH
        LDY     #OMTH
        LDA     (CVARS),Y
        JSR     PRTNUM          ;PRINT THE NUMBER
        LDA     #"/"
        JSR     WRCHAR          ;PRINT A SLASH

        ;PRINT DAY
        LDY     #ODAY
        LDA     (CVARS),Y
        JSR     PRTNUM          ;PRINT THE NUMBER
        LDA     #"/"
        JSR     WRCHAR          ;PRINT A SLASH

        ;PRINT YEAR
        LDY     #OYEAR
        LDA     (CVARS),Y
        SEC
        SBC     CEN20           ;NORMALIZE YEAR TO 20TH CENTURY
        JSR     PRTNUM          ;PRINT THE NUMBER

        ;PRINT SPACE AS DELIMITER
        LDA     #" "
        JSR     WRCHAR          ;PRINT A SPACE BETWEEN DATE AND TIME

        ;PRINT HOURS
        LDY     #OHR
        LDA     (CVARS),Y
        JSR     PRTNUM          ;PRINT THE NUMBER
        LDA     #":"
        JSR     WRCHAR          ;PRINT A COLON

        ;PRINT MINUTES
        LDY     #OMIN
        LDA     (CVARS),Y
        JSR     PRTNUM          ;PRINT THE NUMBER
        LDA     #":"
        JSR     WRCHAR          ;PRINT A COLON

        ;PRINT SECONDS
        LDY     #OSEC
        LDA     (CVARS),Y
        JSR     PRTNUM          ;PRINT THE NUMBER

        ;WAIT UNTIL SECONDS CHANGE THEN PRINT AGAIN
        ;EXIT IF OPERATOR PRESSES A KEY
```

```
        LDY     #OSEC
        LDA     (CVARS),Y
        STA     CURSEC              ;SAVE IN CURRENT SECOND
WAIT:
        ;CHECK KEYBOARD
        JSR     KEYPRS
        BCS     RDKEY               ;BRANCH IF OPERATOR PRESSES A KEY
        LDA     (CVARS),Y           ;GET SECONDS
        CMP     CURSEC
        BEQ     WAIT                ;WAIT UNTIL SECONDS CHANGE
        JMP     LOOP                ;CONTINUE

        ;OPERATOR PRESSED A KEY - DONE IF ESCAPE, PROMPT OTHERWISE
RDKEY:
        JSR     RDCHAR              ;GET CHARACTER
        CMP     #ESC                ;IS IT AN ESCAPE?
        BEQ     DONE                ;BRANCH IF IT IS, ROUTINE IS FINISHED
        JMP     PROMPT              ;ELSE PROMPT OPERATOR FOR NEW STARTING TIME

DONE:
        LDA     #0
        STA     CH                  ;CURSOR TO HORIZONTAL POSITION 0
        LDA     #12
        STA     CV
        JSR     VTAB                ;MOVE CURSOR TO LINE 13 BELOW DISPLAY
        BRK
        JMP     SC1104              ;CONTINUE AGAIN

;********************************
;ROUTINE: KEYPRS
;PURPOSE: DETERMINE IF OPERATOR HAS PRESSED A KEY
;ENTRY: NONE
;EXIT:  IF OPERATOR HAS PRESSED A KEY THEN
;         CARRY = 1
;       ELSE
;         CARRY = 0
;REGISTERS USED: P
;********************************

KEYPRS:
        PHA
        LDA     0C000H              ;READ APPLE KEYBOARD PORT
        ASL     A                   ;MOVE BIT 7 TO CARRY
                                    ; CARRY = 1 IF CHARACTER IS READY ELSE 0
        PLA
        RTS


;********************************
;ROUTINE: RDCHAR
;PURPOSE: READ A CHARACTER
;ENTRY: NONE
;EXIT:  REGISTER A = CHARACTER
;REGISTERS USED: A,P
;********************************
```

```
RDCHAR:
        PHA                         ;SAVE A,X,Y
        TYA
        PHA
        TXA
        PHA

        JSR     RCHAR               ;APPLE MONITOR RDCHAR
        TSX
        AND     #01111111B          ;ZERO BIT 7
        STA     103H,X              ;STORE CHARACTER IN STACK SO IT WILL BE
                                    ; RESTORED TO REGISTER A
        PLA                         ;RESTORE A,X,Y
        TAX
        PLA
        TAY
        PLA
        RTS

;********************************
;ROUTINE: WRCHAR
;PURPOSE: WRITE A CHARACTER
;ENTRY: REGISTER A = CHARACTER
;EXIT:  NONE
;REGISTERS USED: P
;********************************

WRCHAR:
        PHA                         ;SAVE A,X,Y
        TYA
        PHA
        TXA
        PHA

        TSX
        LDA     103H,X              ;GET REGISTER A BACK FROM STACK
        ORA     #10000000B          ;SET BIT 7
        JSR     COUT                ;OUTPUT VIA APPLE MONITOR

        PLA                         ;RESTORE A,X,Y
        TAX
        PLA
        TAY
        PLA
        RTS

;********************************
;ROUTINE: RDLINE
;PURPOSE: READ A LINE TO 200H USING THE APPLE MONITOR
;ENTRY: NONE
;EXIT:  REGISTER X = LENGTH OF LINE
;REGISTERS USED: ALL
;********************************
```

```
RDLINE:
        JSR     GETLN1          ;CALL THE APPLE MONITOR GETLN1
        RTS

;*********************************
;ROUTINE: NXTNUM
;PURPOSE: GET A NUMBER FROM THE INPUT LINE IF ANY
;        IF NONE RETURN A 0
;ENTRY:  LLEN = LENGTH OF THE LINE
;        LIDX = INDEX INTO THE LINE OF NEXT CHARACTER
;EXIT:   REGISTER A = LOW BYTE OF NUMBER
;        REGISTER Y = HIGH BYTE OF NUMBER
;        LIDX = INDEX OF THE FIRST NON NUMERICAL CHARACTER
;REGISTERS USED: ALL
;*********************************

NXTNUM:
        LDA     #0
        STA     NUM
        STA     NUM+1           ;INITIALIZE NUMBER TO 0

        ;WAIT UNTIL A DECIMAL DIGIT IS FOUND (A CHARACTER BETWEEN 30H AND 39H)
        JSR     GETCHR          ;GET NEXT CHARACTER
        BCS     EXITNN          ;EXIT IF END OF LINE
        CMP     #"0"
        BCC     NXTNUM          ;WAIT IF LESS THAN "0"
        CMP     #"9"+1
        BCS     NXTNUM          ;WAIT IF GREATER THAN "9"

        ;FOUND A NUMBER
GETNUM:
        PHA                     ;SAVE CHARACTER ON STACK

        ;MULTIPLY NUM BY TEN
        LDA     NUM
        ASL     A
        ROL     NUM+1
        STA     NUM             ;NUM = LOW BYTE OF NUM * 2
        LDX     NUM+1           ;REGISTER X = HIGH BYTE OF NUM * 2
        ASL     A
        ROL     NUM+1
        ASL     A               ;REGISTER A = LOW BYTE OF NUM * 8
        ROL     NUM+1           ;NUM + 1 = HIGH BYTE OF NUM * 8
        CLC                     ;(NUM * 8) + (NUM * 2) = NUM * 10
        ADC     NUM
        STA     NUM
        TXA
        ADC     NUM+1
        STA     NUM+1

        ;ADD THE CHARACTER TO NUM
        PLA                     ;GET NEXT CHARACTER
        AND     #00001111B      ;NORMALIZE THE CHARACTER TO 0..9
        CLC
        ADC     NUM
        STA     NUM
```

```
        BCC     GETNM1
        INC     NUM+1
GETNM1:

        ;GET THE NEXT CHARACTER
        JSR     GETCHR
        BCS     EXITNN          ;EXIT IF END OF LINE
        CMP     #"0"
        BCC     EXITNN          ;EXIT IF LESS THAN "0"
        CMP     #"9"+1
        BCC     GETNUM          ;STAY IN LOOP IF DIGIT (BETWEEN "0" AND "9")

EXITNN:
        LDA     NUM             ;RETURN THE NUMBER
        LDY     NUM+1
        RTS


;********************************
;ROUTINE: GETCHR
;PURPOSE: GET A CHARACTER FOR THE LINE
;ENTRY: LIDX = NEXT CHARACTER TO GET
;       LLEN = LENGTH OF LINE
;EXIT:  IF NO MORE CHARACTERS THEN
;         CARRY = 1
;       ELSE
;         CARRY = 0
;         REGISTER A = CHARACTER
;REGISTERS USED: ALL
;********************************

GETCHR:
        LDA     LIDX
        CMP     LLEN
        BCS     EXITGC          ;EXIT CHARACTER GET WITH CARRY = 1 TO
                                ; INDICATE END OF LINE (LIDX >= LLEN)
                                ; OTHERWISE, CARRY IS CLEARED

        TAY
        LDA     200H,Y          ;GET CHARACTER
        AND     #01111111B      ;CLEAR BIT 7
        INY                     ;INCREMENT TO NEXT CHARACTER
        STY     LIDX
                                ; CARRY IS STILL CLEARED
EXITGC:
        RTS

;********************************
;ROUTINE: PRTNUM
;PURPOSE: PRINT A NUMBER BETWEEN 0..99
;ENTRY: A = NUMBER TO PRINT
;EXIT:  NONE
;REGISTERS USED: ALL
;********************************
```

```
PRTNUM:
        LDY     #"0"-1          ;INITIALIZE Y TO "0" - 1
        SEC                     ;  Y WILL BE THE 10'S PLACE
DIV10:
        INY                     ;INCREMENT 10'S
        SBC     #10
        BCS     DIV10
        ADC     #10+"0"         ;MAKE REGISTER A AN ASCII DIGIT

        ;REG A = 1'S PLACE
        ;REG Y = 10'S PLACE
        TAX                     ;SAVE 1'S
        TYA
        JSR     WRCHAR          ;OUTPUT 10'S PLACE
        TXA
        JSR     WRCHAR          ;OUTPUT 1'S PLACE
        RTS

;DATA SECTION
CR      .EQU    0DH             ;ASCII CARRIAGE RETURN
MSG     .BYTE   "ENTER DATE AND TIME ",CR,"(MM/DD/YR HR:MN:SC)? ",0
MSGIDX  .BLOCK  1               ;INDEX INTO MESSAGE
NUM:    .BLOCK  2               ;NUMBER
LLEN:   .BLOCK  1               ;LENGTH OF INPUT LINE
LIDX:   .BLOCK  1               ;INDEX OF INPUT LINE
CEN20:  .WORD   1900            ;20TH CENTURY
CURSEC: .BLOCK  1               ;CURRENT SECOND

        .END    ;PROGRAM
```

Appendix A 6502 Instruction Set Summary

Table A-1. 6502 Instructions in Alphabetical Order

INSTRUCTIONS MNEMONIC	OPERATION	IMMEDIATE OP	N	#	ABSOLUTE OP	N	#	ZERO PAGE OP	N	#	ACCUM OP	N	#	IMPLIED OP	N	#	(IND X) OP	N	#	(IND), Y OP	N	#
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2
AND	A ∧ M → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2
ASL	C ◄ [7 0] ◄ Ø				ØE	6	3	Ø6	5	2	ØA	2	1									
6CC	BRANCH ON C=Ø (2)																					
BCS	6RANCH ON C=1 (2)																					
BEO	BRANCH ON Z=1 (2)																					
6IT	A ∧ M				2C	4	3	24	3	2												
BMI	6RANCH ON N=1 (2)																					
6NE	BRANCH ON Z=Ø (2)																					
BPL	6RANCH ON N=Ø (2)																					
BRK	(See Fig. A-1)													ØØ	7	1						
6VC	6RANCH ON V=Ø (2)																					
BVS	BRANCH ON V=1 (2)																					
CLC	Ø → C													18	2	1						
CLO	Ø → O													OB	2	1						
CLI	Ø → I													58	2	1						
CLV	Ø → V													B8	2	1						
CMP	A-M (1)	C9	2	2	CO	4	3	C5	3	2							C1	6	2	O1	5	2
CPX	X-M	EØ	2	2	EC	4	3	E4	3	2												
CPY	Y-M	CØ	2	2	CC	4	3	C4	3	2												
DEC	M-1 → M				CE	6	3	C6	5	2												
DEX	X-1 → X													CA	2	1						
OEY	Y-1 → Y													6B	2	1						
EOR	A ⊻ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	61	6	2
INC	M + 1 → M				EE	6	3	E6	5	2												
INX	X + 1 → X													E6	2	1						
INY	Y + 1 → Y													C8	2	1						
JMP	JUMP TO NEW LOC				4C	3	3															
JSR	(See Fig. A-2) JUMP SU6				2Ø	6	3															
LDA	M → A (1)	A9	2	2	AO	4	3	A5	3	2							A1	6	2	B1	5	2

MNEMONIC	Z PAGE X OP	N	#	ABS, X OP	N	#	ABS Y OP	N	#	RELATIVE OP	N	#	INDIRECT OP	N	#	Z PAGE Y OP	N	#	CONDITION CODES N	Z	C	I	D	V
ADC	75	4	2	7O	4	3	79	4	3										✓	✓	✓	—	—	✓
AND	36	4	2	3D	4	3	39	4	3										✓	✓	—	—	.	—
ASL	16	6	2	1E	7	3													✓	✓	✓	—	—	—
6CC										9Ø	2	2							—	—	—	—	—	—
BCS										BØ	2	2							—	—	—	—	—	—
BEO										FØ	2	2							—	—	—	—	—	—
6IT																			M_7	✓	—	—	—	M_6
BMI										3Ø	2	2							—	—	—	—	—	—
6NE										DØ	2	2							—	—	—	—	—	—
BPL										1Ø	2	2							—	—	—	—	—	—
BRK																			—	—	—	—	—	—
6VC										5Ø	2	2							—	—	—	—	—	—
BVS										7Ø	2	2							—	—	—	—	—	—
CLC																			—	—	Ø	—	—	—
CLO																			—	—	—	—	Ø	—
CLI																			—	—	—	Ø	—	—
CLV																			—	—	—	—	—	Ø
CMP	O5	4	2	DD	4	3	O9	4	3										✓	✓	✓	—	—	—
CPX																			✓	✓	✓	—	—	—
CPY																			✓	✓	✓	—	—	—
DEC	D6	6	2	OE	7	3													✓	✓	—	—	—	—
DEX																			✓	✓	—	—	—	—
OEY																			✓	✓	—	—	—	—
EOR	55	4	2	5D	4	3	59	4	3										✓	✓	—	—	—	—
INC	F6	6	2	FE	7	3													✓	✓	—	—	—	—
INX																			✓	✓	—	—	—	—
INY																			✓	✓	—	—	—	—
JMP													6C	5	3				—	—	—	—	—	—
JSR																			—	—	—	—	—	—
LDA	B5	4	2	BO	4	3	B9	4	3										✓	✓	—	—	—	—

Table A-1. 6502 Instructions in Alphabetical Order (Continued)

		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM.			IMPLIED			(IND, X)			(IND), Y			Z, PAGE, X			ABS, X			ABS, Y			RELATIVE			INDIRECT			Z, PAGE, Y			CONDITION CODES					
MNEMONIC	OPERATION	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N	Z	C	I	D	V
LDX	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	✓	✓	–	–	–	–
LDY	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													✓	✓	–	–	–	–
LSR	0 → [7 0] → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0	✓	✓	–	–	–
NOP	NO OPERATION													EA	2	1																									–	–	–	–	–	–
ORA	A V M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										✓	✓	–	–	–	–
PHA	A → Ms S−1 → S													48	3	1																									–	–	–	–	–	–
PHP	P → Ms S−1 → S													08	3	1																									–	–	–	–	–	–
PLA	S + 1 → S Ms → A													68	4	1																									✓	✓	–	–	–	–
PLP	S + 1 → S Ms → P													28	4	1																									(RESTORED)					
ROL	[7 0] ← [C] ←				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													✓	✓	✓	–	–	–
ROR	→ [C] → [7 0] →				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													✓	✓	✓	–	–	–
RTI	(See Fig. A-1) RTRN INT.													40	6	1																									RESTORED					
RTS	(See Fig. A-2) RTRN SUB													60	6	1																									–	–	–	–	–	–
SBC	A − M − $\bar{C}$ → A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										✓	✓	(3)	–	–	✓
SEC	1 → C													38	2	1																									–	–	1	–	–	–
SED	1 → D													F8	2	1																									–	–	–	–	1	–
SEI	1 → I													78	2	1																									–	–	–	1	–	–
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										–	–	–			
STX	X → M				8E	4	3	86	3	2																												96	4	2	–	–	–	–	–	–
STY	Y → M				8C	4	3	84	3	2													94	4	2																–	–	–	–	–	–
TAX	A → X													AA	2	1																									✓	✓	–	–	–	–
TAY	A → Y													A8	2	1																									✓	✓	–	–	–	–
TSX	S → X													BA	2	1																									✓	✓	–	–	–	–
TXA	X → A													8A	2	1																									✓	✓	–	–	–	–
TXS	X → S													9A	2	1																									–	–	–	–	–	–
TYA	Y → A													98	2	1																									✓	✓	–	–	–	–

(1) ADD 1 TO "N" IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW
(4) IF IN DECIMAL MODE Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
Ms MEMORY PER STACK POINTER

\+ ADD
− SUBTRACT
∧ AND
V OR
⊻ EXCLUSIVE OR
✓ MODIFIED

– NOT MODIFIED
M_7 MEMORY BIT 7
M_6 MEMORY BIT 6
N NUMBER OF CLOCK CYCLES
\# NUMBER OF BYTES

Table A-2. 6502 Operation Codes in Numerical Order

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	LSD / MSD
0	BRK	ORA IND, X		–		ORA Z, Page	ASL Z PAGE		PHP	ORA IMM	ASL A			ORA ABS	ASL ABS		0
1	BPL	ORA IND, Y				ORA Z, Page, X	ASL Z, Page, X		CLC	ORA ABS, Y				ORA ABS, X	ASL ABS, X		1
2	JSR	AND IND, X			BIT Z, Page	AND Z, Page	ROL Z Page		PLP	AND IMM	ROL A		BIT ABS	AND ABS	ROL ABS		2
3	BMI	AND IND, Y				AND Z, Page, X	ROL Z, Page, X		SEC	AND ABS, Y				AND ABS, X	ROL ABS, X		3
4	RTI	EOR IND, X				EOR Z, Page	LSR Z, Page		PHA	EOR IMM	LSR A		JMP ABS	EOR ABS	LSR ABS		4
5	BVC	EOR IND, Y				EOR Z, Page, X	LSR Z, Page, X		CLI	EOR ABS, Y				EOR ABS, X	LSR ABS, X		5
6	RTS	ADC IND, X				ADC Z, Page	ROR Z, Page		PLA	ADC IMM	ROR A		JMP IND	ADC ABS	ROR ABS		6
7	BVS	ADC IND, Y				ADC Z, Page, X			SEI	ADC ABS, Y				ADC ABS, X			7
8		STA IND, X			STY Z, Page	STA Z, Page	STX Z, Page		DEY		TXA		STY ABS	STA ABS	STX ABS		8
9	BCC	STA IND, Y			STY Z, Page, X	STA Z, Page, X	STX Z, Page, Y		TYA	STA ABS, Y	TXS			STA ABS, X			9
A	LDY IMM	LDA IND, X	LDX IMM		LDY Z, Page	LDA Z, Page	LDX Z, Page		TAY	LDA IMM	TAX		LDY ABS	LDA ABS	LDX ABS		A
B	BCS	LDA IND, Y			LDY Z, Page, X	LDA Z, Page, X	LDX Z, Page, Y		CLV	LDA ABS, Y	TSX		LDY ABS, X	LDA ABS, X	LDX ABS, Y		B
C	CPY IMM	CMP IND, X			CPY Z, Page	CMP Z, Page	DEC Z, Page		INY	CMP IMM	DEX		CPY ABS	CMP ABS	DEC ABS		C
D	BNE	CMP IND, Y				CMP Z, Page, X	DEC Z, Page, X		CLD	CMP ABS, Y				CMP ABS, X	DEC ABS, X		D
E	CPX IMM	SBC IND, X			CPX Z, Page	SBC Z, Page	INC Z, Page		INX	SBC IMM	NOP		CPX ABS	SBC ABS	INC ABS		E
F	BEQ	SBC IND, Y				SBC Z, Page, X	INC Z, Page, X		SED	SBC ABS, Y				SBC ABS, X	INC ABS, X		F

Table A-3. Summary of 6502 Addressing Modes

IMM - IMMEDIATE ADDRESSING – THE OPERAND IS CONTAINED IN THE SECOND BYTE OF THE INSTRUCTION.

ABS - ABSOLUTE ADDRESSING – THE SECOND BYTE OF THE INSTRUCTION CONTAINS THE 8 LOW ORDER BITS OF THE EFFECTIVE ADDRESS. THE THIRD BYTE CONTAINS THE 8 HIGH ORDER BITS OF THE EFFECTIVE ADDRESS.

Z, PAGE - ZERO PAGE ADDRESSING – SECOND BYTE CONTAINS THE 8 LOW ORDER BITS OF THE EFFECTIVE ADDRESS. THE 8 HIGH ORDER BITS ARE ZERO.

A, - ACCUMULATOR – ONE BYTE INSTRUCTION OPERATING ON THE ACCUMULATOR.

Z, PAGE, X - Z PAGE, Y - ZERO PAGE INDEXED – THE SECOND BYTE OF THE INSTRUCTION IS ADDED TO THE INDEX (CARRY IS DROPPED) TO FORM THE LOW ORDER BYTE OF THE EA. THE HIGH ORDER BYTE OF THE EA IS ZERO.

ABS, X ABS, Y ABSOLUTE INDEXED – THE EFFECTIVE ADDRESS IS FORMED BY ADDING THE INDEX TO THE SECOND AND THIRD BYTE OF THE INSTRUCTION.

(IND, X) - INDEXED INDIRECT – THE SECOND BYTE OF THE INSTRUCTION IS ADDED TO THE X INDEX, DISCARDING THE CARRY. THE RESULTS POINTS TO A LOCATION ON PAGE ZERO WHICH CONTAINS THE 8 LOW ORDER BITS OF THE EA. THE NEXT BYTE CONTAINS THE 8 HIGH ORDER BITS.

(IND), Y - INDIRECT INDEXED – THE SECOND BYTE OF THE INSTRUCTION POINTS TO A LOCATION IN PAGE ZERO. THE CONTENTS OF THIS MEMORY LOCATION IS ADDED TO THE Y INDEX, THE RESULT BEING THE LOW ORDER EIGHT BITS OF THE EA. THE CARRY FROM THIS OPERATION IS ADDED TO THE CONTENTS OF THE NEXT PAGE ZERO LOCATION, THE RESULTS BEING THE 8 HIGH ORDER BITS OF THE EA.

Table A-4. 6502 Assembler Directives, Labels, and Special Characters

ASSEMBLER DIRECTIVES

- OPT – SPECIFIES OPTIONS FOR ASSEMBLY
  OPTIONS ARE: (OPTIONS LISTED FIRST ARE THE DEFAULT VALUES).
  NOC (COU OR CNT) – DO NOT LIST ALL INSTRUCTIONS AND THEIR USAGE.
  NOG (GEN) – DO NOT GENERATE MORE THAN ONE LINE OF CODE FOR ASCII STRINGS.
  XRE (NOX) – PRODUCE A CROSS-REFERENCE LIST IN THE SYMBOL TABLE.
  ERR (NOE) – CREATE AN ERROR FILE.
  MEM (NOM) – CREATE AN ASSEMBLER OBJECT OUTPUT FILE.
  LIS (NOL) – PRODUCE A FULL ASSEMBLY LISTING.
- BYTE – PRODUCES A SINGLE BYTE IN MEMORY EQUAL TO EACH OPERAND SPECIFIED.
- WORD – PRODUCES AN ADDRESS (2 BYTES) IN MEMORY EQUAL TO EACH OPERAND SPECIFIED.
- DBYTE – PRODUCES TWO BYTES IN MEMORY EQUAL TO EACH OPERAND SPECIFIED.
- SKIP – GENERATE THE NUMBER OF BLANK LINES SPECIFIED BY THE OPERAND.
- PAGE – ADVANCE THE LISTING TO THE TOP OF A NEW PAGE AND CHANGE TITLE.
- END – DEFINES THE END OF A SOURCE PROGRAM.

* = – DEFINES THE BEGINNING OF A NEW PROGRAM COUNTER SEQUENCE.

LABELS

LABELS ARE THE FIRST FIELD AND MUST BE FOLLOWED BY AT LEAST ONE SPACE OR A COLON (:)
LABELS CAN BE UP TO 6 ALPHANUMERIC CHARACTERS LONG AND MUST BEGIN WITH AN ALPHA CHARACTER.
A,X,Y,S,P AND THE 56 OPCODES ARE RESERVED AND CANNOT BE USED AS LABELS.
LABEL = EXPRESSION CAN BE USED TO EQUATE LABELS TO VALUES.
LABEL *=* +N CAN BE USED TO RESERVE AREAS IN MEMORY.

CHARACTERS USED AS SPECIAL PREFIXES:

- • INDICATES AN ASSEMBLER DIRECTIVE
- \# SPECIFIES THE IMMEDIATE MODE OF ADDRESSING
- $ SPECIFIES A HEXADECIMAL NUMBER
- @ SPECIFIES AN OCTAL NUMBER
- % SPECIFIES A BINARY NUMBER
- ' SPECIFIES AN ASCII LITERAL CHARACTER
- ( ) INDICATES INDIRECT ADDRESSING
- ; INDICATES FOLLOWING TEXT ARE COMMENTS
- < SPECIFIES LOWER HALF OF A 16 BIT VALUE
- > SPECIFIES UPPER HALF OF A 16 BIT VALUE

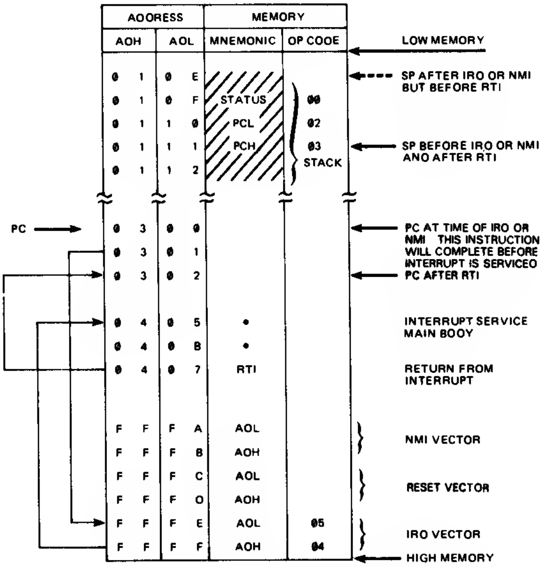


Figure A-1. Response to $\overline{IRQ}$ and $\overline{NMI}$ Inputs and Operation of the RTI and BRK Instructions

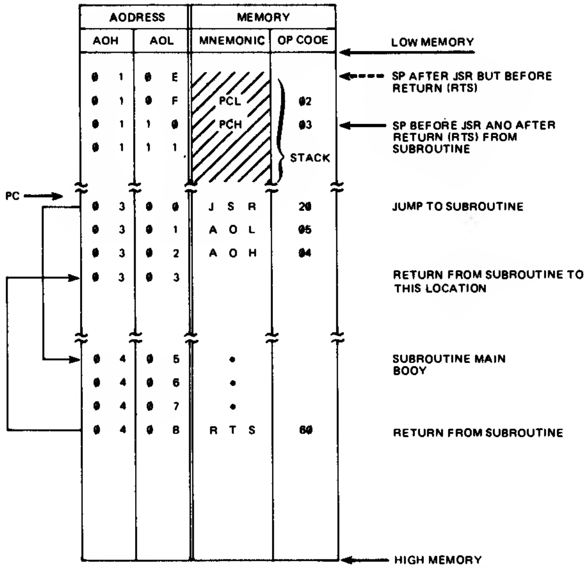


Figure A-2. Operation of the JSR and RTS Instructions

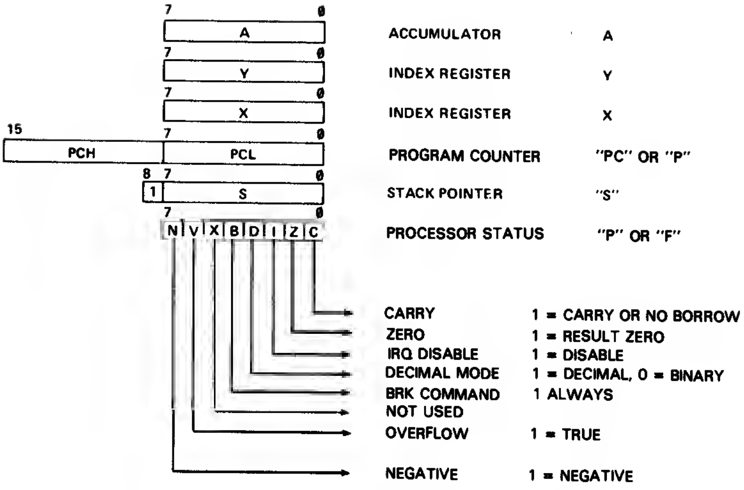


Figure A-3. Programming Model of the 6502 Microprocessor

Appendix B Programming Reference for the 6522 Versatile Interface Adapter (VIA)

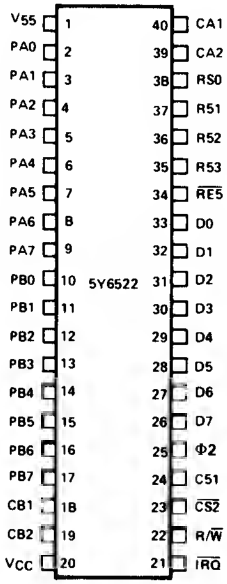


Figure B-1. 6522 Pin Assignments

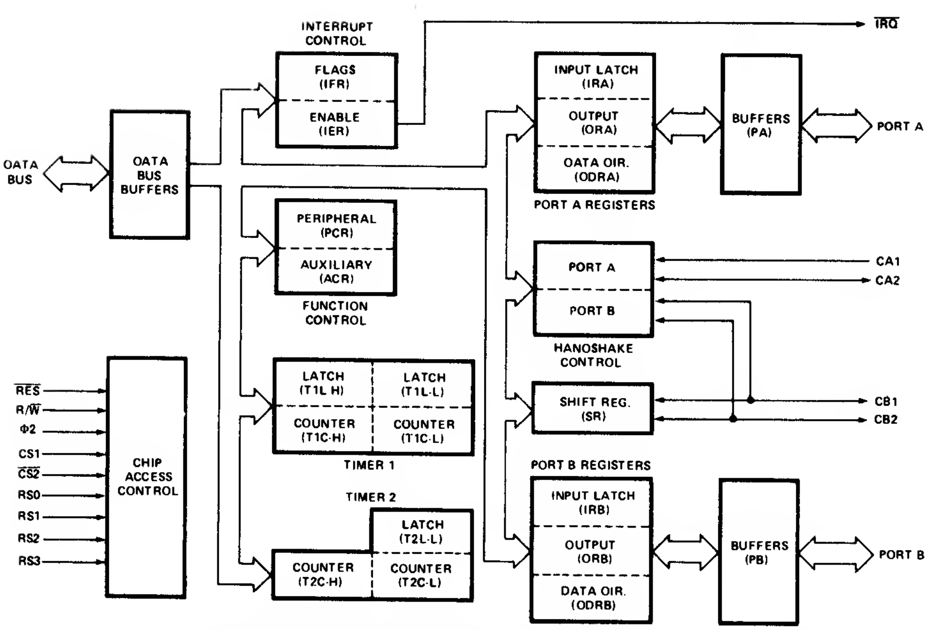
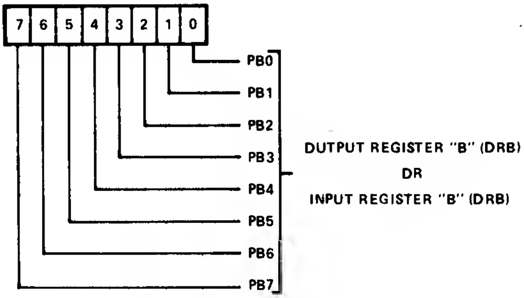


Figure B-2. Block Diagram of the 6522 Versatile Interface Adapter (VIA)

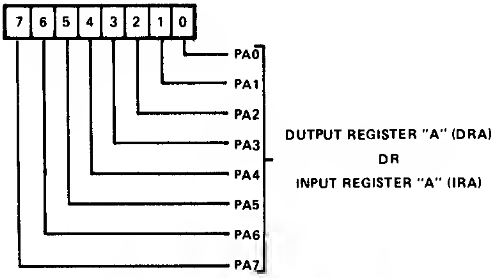
Table B-1. 6522 Internal Registers

Register Number	RS Coding				Register Desig.	Description	
	RS3	RS2	RS1	RS0		Write	Read
0	0	0	0	0	ORB/IRB	Output Register "B"	Input Register "B"
1	0	0	0	1	ORA/IRA	Output Register "A"	Input Register "A"
2	0	0	1	0	DDRB	Data Direction Register "B"	
3	0	0	1	1	DDRA	Data Direction Register "A"	
4	0	1	0	0	T1C-L	T1 Low-Order Latches	T1 Low-Order Counter
5	0	1	0	1	T1C-H	T1 High-Order Counter	
6	0	1	1	0	T1L-L	T1 Low-Order Latches	
7	0	1	1	1	T1L-H	T1 High-Order Latches	
B	1	0	0	0	T2C-L	T2 Low-Order Latches	T2 Low-Order Counter
9	1	0	0	1	T2C-H	T2 High-Order Counter	
10	1	0	1	0	SR	Shift Register	
11	1	0	1	1	ACR	Auxiliary Control Register	
12	1	1	0	0	PCR	Peripheral Control Register	
13	1	1	0	1	IFR	Interrupt Fleg Register	
14	1	1	1	0	IER	Interrupt Enable Register	
15	1	1	1	1	ORA/IRA	Same es Reg 1 Except No "Handshake"	



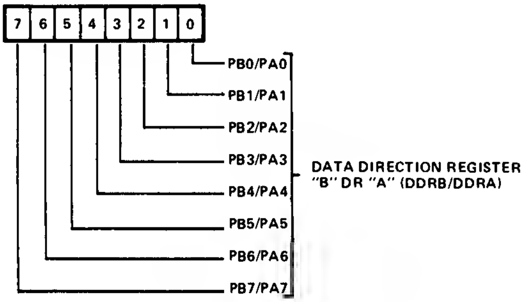
Pin Data Direction Selection	WRITE	READ
DDRB = "1" (OUTPUT)	MPU writes Output Level (DRB)	MPU reads output register bit in ORB. Pin level has no affect.
DDRB = "0" (INPUT) (Input latching disabled)	MPU writes into DRB, but no effect on pin level, until DDRB changed.	MPU reads input level on PB pin.
DDRB = "0" (INPUT) (Input latching enabled)		MPU reads IRB bit, which is the level of the PB pin at the time of the last CB1 active transition.

Figure B-3. Output Register B and Input Register B (Register 0)



Pin Data Direction Selection	WRITE	READ
DDRA = "1" (OUTPUT) (Input latching disabled)	MPU writes Output Level (DRA).	MPU reads level on PA pin.
DDRA = "1" (OUTPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition.
DDRA = "0" (INPUT) (Input latching disabled)	MPU writes into DRA, but no effect on pin level, until DDRA changed.	MPU reads level on PA pin.
DDRA = "0" (INPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition.

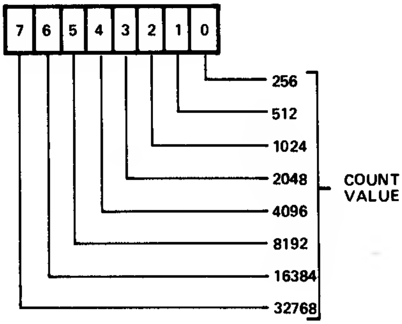
Figure B-4. Output Register A and Input Register A (Register 1)



"0" ASSDCIATED PB/PA PIN IS AN INPUT (HIGH-IMPEDANCE)

"1" ASSDCIATED PB/PA PIN IS AN OUTPUT, WHDSE LEVEL IS DETERMINED BY ORB/DRA REGISTER BIT.

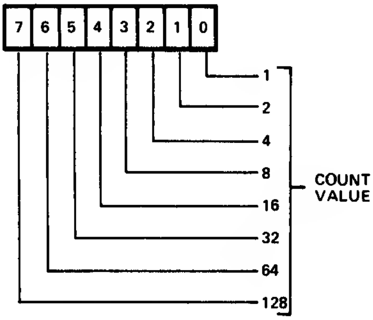
Figure B-5. Data Direction Registers B (Register 2) and A (Register 3)



WRITE – 8 BITS LOADED INTO T1 HIGH-ORDER LATCHES. ALSO, AT THIS TIME BOTH HIGH AND LOW-ORDER LATCHES TRANSFERREO INTO T1 COUNTER. T1 INTERRUPT FLAG ALSO IS RESET.

READ – 8 BITS FROM T1 HIGH-OROER COUNTER TRANSFERREO TO MPU.

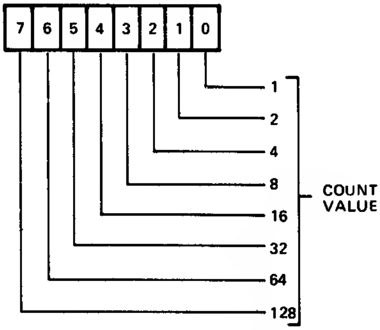
Figure B-7. Timer 1 High-Order Counter (Register 5)



WRITE – 8 BITS LOAOEO INTO T1 LOW-ORDER LATCHES. LATCH CONTENTS ARE TRANSFERREO INTO LOW-ORDER COUNTER AT THE TIME THE HIGH-OROER COUNTER IS LOADED (REG 5).

READ – 8 BITS FROM T1 LOW-ORDER COUNTER TRANSFERRED TO MPU. IN AOOITION, T1 INTERRUPT FLAG IS RESET (BIT 6 IN INTERRUPT FLAG REGISTER).

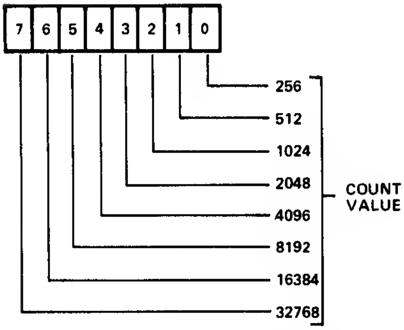
Figure B-6. Timer 1 Low-Order Counter (Register 4)



WRITE – 8 BITS LOADED INTO T1 LOW-ORDER LATCHES. THIS OPERATION IS THE SAME AS WRITING INTO REGISTER 4.

READ – 8 BITS FROM T1 LOW-OROER LATCHES TRANSFERRED TO MPU. UNLIKE REG 4 OPERATION, THIS DOES NOT CAUSE RESET OF T1 INTERRUPT FLAG.

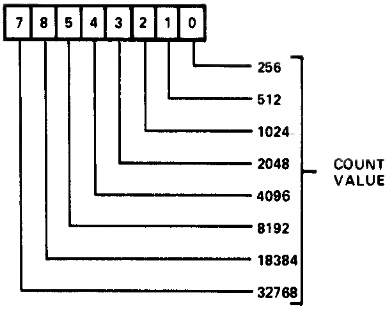
Figure B-8. Timer 1 Low-Order Latches (Register 6)



WRITE – 8 BITS LOADED INTO T1 HIGH-ORDER LATCHES. UNLIKE REG 4 OPERATION NO LATCH-TO-COUNTER TRANSFERS TAKE PLACE.

READ – 8 BITS FROM T1 HIGH-ORDER LATCHES TRANSFERRED TO MPU.

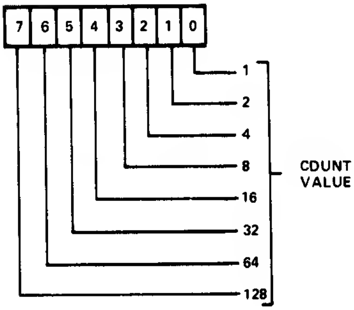
Figure B-9. Timer 1 High-Order Latches (Register 7)



WRITE – 8 BITS LOADED INTO T2 HIGH-ORDER COUNTER. ALSO, LOW-ORDER LATCHES TRANSFERRED TO LOW-ORDER COUNTER. IN ADDITION, T2 INTERRUPT FLAG IS RESET.

READ – 8 BITS FROM T2 HIGH-ORDER COUNTER TRANSFERRED TO MPU.

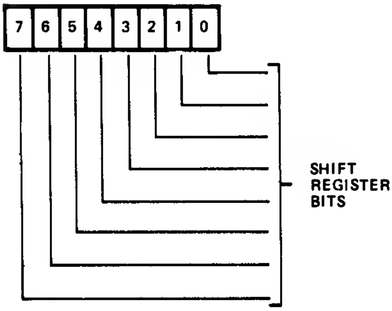
Figure B-11. Timer 2 High-Order Counter (Register 9)



WRITE – 8 BITS LOADED INTO T2 LOW-ORDER LATCHES.

READ – 8 BITS FROM T2 LOW-ORDER COUNTER TRANSFERRED TO MPU. T2 INTERRUPT FLAG IS RESET.

Figure B-10. Timer 2 Low-Order Counter (Register 8)



NOTES:
1. WHEN SHIFTING OUT, BIT 7 IS THE FIRST BIT OUT AND SIMULTANEOUSLY IS ROTATED BACK INTO BIT 0.
2. WHEN SHIFTING IN, BITS INITIALLY ENTER BIT 0 AND ARE SHIFTED TOWARDS BIT 7.

Figure B-12. Shift Register (Register 10)

7	6	5	4	3	2	1	0

T1 TIMER CONTROL

7	6	OPERATION	PB7
0	0	TIMEO INTERRUPT EACH TIME T1 IS LOAOEO	DISABLED
0	1	CONTINUOUS INTERRUPTS	
1	0	TIMED INTERRUPT EACH TIME T1 IS LOAOEO	ONE-SHOT OUTPUT
1	1	CONTINUOUS INTERRUPTS	SOUARE WAVE OUTPUT

T2 TIMER CONTROL

5	OPERATION
0	TIMEO INTERRUPT
1	COUNT OOWN WITH PULSES ON PB6

SHIFT REGISTER CONTROL

4	3	2	OPERATION
0	0	0	OISABLEO
0	0	1	SHIFT IN UNOER CONTROL OF T2
0	1	0	SHIFT IN UNOER CONTROL OF $\phi 2$
0	1	1	SHIFT IN UNOER CONTROL OF EXT. CLK
1	0	0	SHIFT OUT FREE-RUNNING AT T2 RATE
1	0	1	SHIFT OUT UNOER CONTROL OF T2
1	1	0	SHIFT OUT UNOER CONTROL OF $\phi 2$
1	1	1	SHIFT OUT UNOER CONTROL OF EXT. CLK.

LATCH ENABLE/OISABLE

PA

PB

0 = OISABLE
1 = ENABLE LATCHING

Figure B-13. Auxiliary Control Register (Register 11)

7	6	5	4	3	2	1	0

CB2 CONTROL

7	6	5	OPERATION
0	0	0	INPUT-NEGATIVE ACTIVE EDGE
0	0	1	INDEPENDENT INTERRUPT INPUT-NEG EDGE
0	1	0	INPUT-PDSITIVE ACTIVE EDGE
0	1	1	INDEPENDENT INTERRUPT INPUT-POS EDGE
1	0	0	HANDSHAKE OUTPUT
1	0	1	PULSE OUTPUT
1	1	0	LOW OUTPUT
1	1	1	HIGH OUTPUT

CB1 INTERRUPT CONTRDL

0 = NEGATIVE ACTIVE EDGE
1 = POSITIVE ACTIVE EDGE

CA1 INTERRUPT CONTROL

0 = NEGATIVE ACTIVE EDGE
1 = POSITIVE ACTIVE EDGE

CA2 CONTROL

3	2	1	DPERATIDN
0	0	0	INPUT-NEGATIVE ACTIVE EDGE
0	0	1	INDEPENDENT INTERRUPT INPUT-NEG EDGE
0	1	0	INPUT-POSITIVE ACTIVE EDGE
0	1	1	INDEPENDENT INTERRUPT INPUT-POS EDGE
1	0	0	HANDSHAKE OUTPUT
1	0	1	PULSE OUTPUT
1	1	0	LOW OUTPUT
1	1	1	HIGH OUTPUT

Figure B-14. Peripheral Control Register (Register 12)

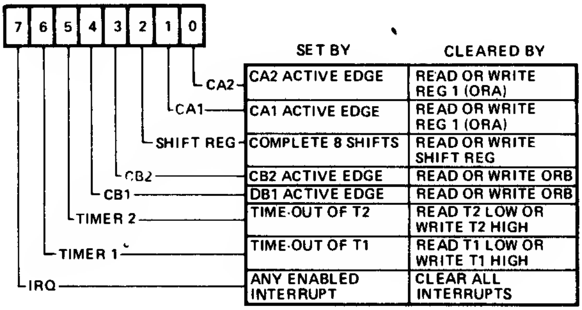
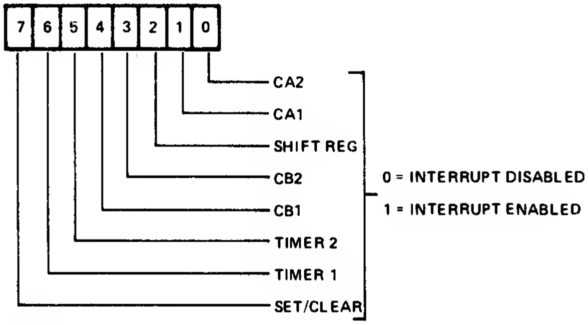


Figure B-15. Interrupt Flag Register (Register 13)



NOTES:
1. IF BIT 7 IS A "0", THEN EACH "1" IN BITS 0 - 6 DISABLES THE CORRESPONDING INTERRUPT.
2. IF BIT 7 IS A "1", THEN EACH "1" IN BITS 0 - 6 ENABLES THE CORRESPONDING INTERRUPT.
3. IF A READ OF THIS REGISTER IS DONE, BIT 7 WILL BE "1" AND ALL OTHER BITS WILL REFLECT THEIR ENABLE/DISABLE STATE.

Figure B-16. Interrupt Enable Register (Register 14)

Appendix C **ASCII Character Set**

LSD \ MSD		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENG	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	}
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	{
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	VS	/	?	O	—	o	DEL

Glossary

A

Absolute address. An address that identifies a storage location or a device without the use of a base, offset, or other factor. *See also* Effective address, Relative offset.

Absolute addressing. An addressing mode in which the instruction contains the actual address required for its execution. In 6502 terminology, absolute addressing refers to a type of direct addressing in which the instruction contains a full 16-bit address as opposed to zero page addressing in which the instruction contains only an 8-bit address on page 0.

Absolute indexed addressing. A form of indexed addressing in which the instruction contains a full 16-bit base address.

Accumulator. A register that is the implied source of one operand and the destination of the result for most arithmetic and logic operations.

ACIA (Asynchronous Communications Interface Adapter). A serial interface device. Common ACIAs in 6502-based computers are the 6551 and 6850 devices. *See also* UART.

Active transition (in a PIA or VIA). The edge on the control line that sets an Interrupt flag. The alternatives are a negative edge (1 to 0 transition) or a positive edge (0 to 1 transition).

Address. The identification code that distinguishes one memory location or input/output port from another and that can be used to select a specific one.

Addressing modes. The methods for specifying the addresses to be used in executing an instruction. Common addressing modes are direct, immediate, indexed, indirect, and relative.

Address register. A register that contains a memory address.

Address space. The total range of addresses to which a particular computer may refer.

ALU. See Arithmetic-logic unit.

Arithmetic-logic unit (ALU). A device that can perform any of a variety of arithmetic or logical functions; function inputs select which function is performed during a particular cycle.

Arithmetic shift. A shift operation that preserves the value of the sign bit (most significant bit). In a right shift, this results in the sign bit being copied into the succeeding bit positions (called *sign extension*).

Arm. See Enable, but most often applied to interrupts.

Array. A collection of related data items, usually stored in consecutive memory addresses.

ASCII (American Standard Code for Information Interchange). A 7-bit character code widely used in computers and communications.

Assembler. A computer program that converts assembly language programs into a form (machine language) that the computer can execute directly. The assembler translates mnemonic operation codes and names into their numerical equivalents and assigns locations in memory to data and instructions.

Assembly language. A computer language in which the programmer can use mnemonic operation codes, labels, and names to refer to their numerical equivalents.

Asynchronous. Operating without reference to an overall timing source, that is, at irregular intervals.

Autodecrementing. The automatic decrementing of an address register as part of the execution of an instruction that uses it.

Autoincrementing. The automatic incrementing of an address register as part of the execution of an instruction that uses it.

Automatic mode (of a peripheral chip). An operating mode in which the peripheral chip produces control signals automatically without specific program intervention.

B

Base address. The address in memory at which an array or table starts. Also called *starting address* or *base.*

Baud. A measure of the rate at which serial data is transmitted, bits per second, but including both data bits and bits used for synchronization, error checking, and other purposes. Common baud rates are 110, 300, 1200, 2400, 4800, and 9600.

Baud rate generator. A device that generates the proper time intervals between bits for serial data transmission.

BCD (Binary-Coded Decimal). A representation of decimal numbers in which each decimal digit is coded separately into a binary number.

Bidirectional. Capable of transporting signals in either direction.

Binary-coded decimal. See BCD.

Binary search. A search in which the set of items to be searched is divided into two equal (or nearly equal) parts during each iteration. The part containing the item being sought is then determined and used as the set in the next iteration. A binary search thus halves the size of the set being searched with each iteration. This method obviously assumes the set of items is ordered.

Bit test. An operation that determines whether a bit is 0 or 1. Usually refers to a logical AND operation with an appropriate mask.

Block. An entire group or section, such as a set of registers or a section of memory.

Block comparison (or block compare). A search that extends through a block of memory until either the item being sought is found or the entire block is examined.

Block move. Moving an entire set of data from one area of memory to another.

Boolean variable. A variable that has only two possible values, which may be represented as true and false or as 1 and 0. *See also* Flag.

Borrow. A bit which is set to 1 if a subtraction produces a negative result and to 0 if it produces a positive or zero result. The borrow is used commonly to subtract numbers that are too long to be handled in a single operation.

Bounce. To move back and forth between states before reaching a final state.

Branch instruction. See Jump instruction.

Break instruction. See Trap.

Breakpoint. A condition specified by the user under which program execution is to end temporarily. Breakpoints are used as an aid in debugging. The specification of the conditions under which execution will end is referred to as *setting*

breakpoints and the deactivation of those conditions is referred to as *clearing breakpoints*.

BSC (Binary Synchronous Communications or BISYNC). An older line protocol often used by IBM computers and terminals.

Bubble sort. A sorting technique which goes through an array exchanging each pair of elements that are out of order.

Buffer. Temporary storage area generally used to hold data before it is transferred to its final destination.

Buffer empty. A signal that is active when any data entered into a buffer or register has been transferred to its final destination.

Buffer full. A signal that is active when a buffer or register is completely occupied with data that has not been transferred to its final destination.

Buffer index. The index of the next available address in a buffer.

Buffer pointer. A storage location that contains the next available address in a buffer.

Bug. An error or flaw.

Byte. A unit of eight bits. May be described as consisting of a high nibble or digit (the four most significant bits) and a low nibble or digit (the four least significant bits).

Byte-length. A length of eight bits per item.

C

Call (a subroutine). Transfers control to the subroutine while retaining the information required to resume the current program. A call differs from a jump or branch in that a call retains information concerning its origin, whereas a jump or branch does not.

Carry. A bit that is 1 if an addition overflows into the succeeding digit position.

Carry flag. A flag that is 1 if the last operation generated a carry from the most significant bit and 0 if it did not.

CASE statement. A statement in a high-level computer language that directs the computer to perform one of several subprograms, depending on the value of a variable. That is, the computer performs the first subprogram if the variable has the first value specified, etc. The computed GO TO statement serves a similar function in FORTRAN.

Central processing unit (CPU). The control section of the computer which controls its operations, fetches and executes instructions, and performs arithmetic and logical functions.

Checksum. A logical sum that is included in a block of data to guard against recording or transmission errors. Also referred to as *longitudinal parity* or *longitudinal redundancy check* (LRC).

Circular shift. See Rotate.

Cleaning the stack. Removing unwanted items from the stack, usually by adjusting the stack pointer.

Clear. Set to zero.

Clock. A regular timing signal that governs transitions in a system.

Close (a file). To make a file inactive. The final contents of the file are the last information the user stored in it. The user must generally close a file after working with it.

Coding. Writing instructions in a computer language.

Combo chip. See Multifunction device.

Command register. See Control register.

Comment. A section of a program that has no function other than documentation. Comments are neither translated nor executed, but are simply copied into the program listing.

Complement. Invert; *see also* one's complement, two's complement.

Concatenation. Linking together, chaining, or uniting in a series. In string operations, placing of one string after another.

Condition code. See Flag.

Control (command) register. A register whose contents determine the state of a transfer or the operating mode of a device.

Control signal. A signal that directs an I/O transfer or changes the operating mode of a peripheral.

Cyclic redundancy check (CRC). An error-detecting code generated from a polynomial that can be added to a block of data or a storage area.

D

Data accepted. A signal that is active when the most recent data has been transferred successfully.

Data direction register. A register that determines whether bidirectional I/O lines are being used as inputs or outputs. Abbreviated as DDR in some diagrams.

Data-link control. A set of conventions governing the format and timing of data exchange between communicating systems. Also called a *protocol.*

Data ready. A signal that is active when new data is available to the receiver. Same as *valid data.*

Data register. In a PIA or VIA, the actual input/output port. Also called an *output register* or a *peripheral register.*

DDCMP (Digital Data Communications Message Protocol). A widely used protocol that supports any method of physical data transfer (synchronous or asynchronous, serial or parallel).

Debounce. Convert the output from a contact with bounce into a single, clean transition between states. Debouncing is most commonly applied to outputs from mechanical keys or switches which bounce back and forth before settling into their final positions.

Debounce time. The amount of time required to debounce a change of state.

Debugger. A program that helps in locating and correcting errors in a user program. Some versions are referred to as dynamic debugging tools or DDT after the famous insecticide.

Debugging. The process of locating and correcting errors in a program.

Device address. The address of a port associated with an input or output device.

Diagnostic. A program that checks the operation of a device and reports its findings.

Digit shift. A shift of one BCD digit position or four bit positions.

Direct addressing. An addressing mode in which the instruction contains the address required for its execution. The 6502 microprocessor has two types of direct addressing: zero page addressing (requiring only an 8-bit address on page 0) and absolute addressing (requiring a full 16-bit address in two bytes of memory).

Disarm. See Disable, but most often applied to interrupts.

Disable (or disarm). Prohibit an activity from proceeding or a signal (such as an interrupt) from being recognized.

Double word. A unit of 32 bits.

Driver. See I/O driver.

Dump. A facility that displays the contents of an entire section of memory or group of registers on an output device.

Dynamic allocation (of memory). The allocation of memory for a subprogram from whatever is available when the subprogram is called. This is as opposed to the *static allocation* of a fixed area of storage to each subprogram. Dynamic allocation often reduces memory usage because subprograms can share areas; it does, however, generally require additional execution time and overhead spent in memory management.

E

EBCDIC (Expanded Binary-Coded Decimal Interchange Code). An 8-bit character code often used in large computers.

Echo. Reflects transmitted information back to the transmitter; sends back to a terminal the information received from it.

Editor. A program that manipulates text material and allows the user to make corrections, additions, deletions, and other changes.

Effective address. The actual address used by an instruction to fetch or store data.

EIA RS-232. See RS-232.

Enable (or arm). Allows an activity to proceed or a signal (such as an interrupt) to be recognized.

Endless loop or *jump-to-self instruction.* An instruction that transfers control to itself, thus executing indefinitely (or until a hardware signal interrupts it).

Error-correcting code. A code that the receiver can use to correct errors in messages; the code itself does not contain any additional message.

Error-detecting code. A code that the receiver can use to detect errors in messages; the code itself does not contain any additional message.

Even parity. A 1-bit error-detecting code that makes the total number of 1 bits in a unit of data (including the parity bit) even.

EXCLUSIVE OR function. A logical function that is true if either of its inputs is true but not both. It is thus true if its inputs are not equal (that is, if one of them is a logic 1 and the other is a logic 0).

External reference. The use in a program of a name that is defined in another program.

F

F (flag) register. See Processor status register.

File. A collection of related information that is treated as a unit for purposes of storage or retrieval.

Fill. Placing values in storage areas not previously in use, initializing memory or storage.

Flag (or condition code or status bit). A single bit that indicates a condition within the computer, often used to choose between alternative instruction sequences.

Flag (software). An indicator that is either on (1) or off (0) and can be used to select between two alternative courses of action. *Boolean variable* and *semaphore* are other terms with the same meaning.

Flag register. See Processor status register.

Free-running mode. An operating mode for a timer in which it indicates the end of a time interval and then starts another of the same length. Also referred to as a *continuous mode.*

Function key. A key that causes a system to perform a function (such as clearing the screen of a video terminal) or execute a procedure.

G

Global. This is a universal variable. Defined in more than one section of a computer program, rather than used only locally.

H

Handshake. An asynchronous transfer in which sender and receiver exchange predetermined signals to establish synchronization and to indicate the status of the data transfer. Typically, the sender indicates that new data is available and the receiver reads the data and indicates that it is ready for more.

Hardware stack. A stack that the computer automatically manages when executing instructions that use it.

Head (of a queue). The location of the item most recently entered into the queue.

Header, queue. See Queue header.

Hexadecimal (or hex). Number system with base 16. The digits are the decimal numbers 0 through 9, followed by the letters A through F.

Hex code. See Object code.

High-level language. A programming language that is aimed toward the solution of problems, rather than being designed for convenient conversion into computer instructions.A compiler or interpreter translates a program written in a high-level language into a form that the computer can execute. Common high-level languages include BASIC, COBOL, FORTRAN, and Pascal.

I

Immediate addressing. An addressing mode in which the data required by an instruction is part of the instruction. The data immediately follows the operation code in memory.

Independent mode (of a parallel interface). An operating mode in which the status and control signals associated with a parallel I/O port can be used independently of data transfers through the port.

Index. A data item used to identify a particular element of an array or table.

Indexed addressing. An addressing mode in which the address is modified by the contents of an index register to determine the effective address (the actual address used).

Indexed indirect addressing. An addressing mode in which the effective address is determined by indexing from the base address and then using the indexed address indirectly. This is also known as *preindexing,* since the indexing is performed before the indirection. Of course, the array starting at the given base address must consist of addresses that can be used indirectly.

Index register. A register that can be used to modify memory addresses.

Indirect addressing. An addressing mode in which the effective address is the contents of the address included in the instruction, rather than the address itself.

Indirect indexed addressing. An addressing mode in which the effective address is determined by first obtaining the base address indirectly and then indexing from that base address. Also known as *postindexing,* since the indexing is performed after the indirection.

Indirect jump. A jump instruction that transfers control to the address stored in a register or memory location, rather than to a fixed address.

Input/output control block (IOCB). A group of storage locations that contain the information required to control the operation of an I/O device. Typically included in the information are the addresses of routines that perform operations such as transferring a single unit of data or determining device status.

Input/output control system (IOCS). A set of computer routines that control the performance of I/O operations.

Instruction. A group of bits that defines a computer operation and is part of the instruction set.

Instruction cycle. The process of fetching, decoding, and executing an instruction.

Instruction execution time. The time required to fetch, decode, and execute an instruction.

Instruction fetch. The process of addressing memory and reading an instruction into the CPU for decoding and execution.

Instruction length. The amount of memory needed to store a complete instruction.

Instruction set. The set of general-purpose instructions available on a given computer. The set of inputs to which the CPU will produce a known response when they are fetched, decoded, and executed.

Interpolation. Estimating values of a function at points between those at which the values are already known.

Interrupt. A signal that temporarily suspends the computer's normal sequence of operations and transfers control to a special routine.

Interrupt-driven. Dependent on interrupts for its operation, may idle until it receives an interrupt.

Interrupt flag. A bit in the input/output section that is set when an event occurs that requires servicing by the CPU. Typical events include an active transition on a control line and the exhaustion of a count by a timer.

Interrupt mask (or interrupt enable). A bit that determines whether interrupts will be recognized. A mask or disable bit must be cleared to allow interrupts, whereas an enable bit must be set.

Interrupt request. A signal that is active when a peripheral is requesting service, often used to cause a CPU interrupt. *See also* Interrupt flag.

Interrupt service routine. A program that performs the actions required to respond to an interrupt.

Inverted borrow. A bit which is set to 0 if a subtraction produces a negative result and to 1 if it produces a positive or 0 result. An inverted borrow can be used like a true borrow, except that the complement of its value (i.e., 1 minus its value) must be used in the extension to longer numbers.

IOCB. See Input/output control block.

IOCS. See Input/output control system.

I/O device table. A table that establishes the correspondence between the logical devices to which programs refer and the physical devices that are actually used in data transfers. An I/O device table must be placed in memory in order to run a program that refers to logical devices on a computer with a particular set of actual (physical) devices. The I/O device table may, for example, contain the starting addresses of the I/O drivers that handle the various devices.

I/O driver. A computer program that transfers data to or from an I/O device, also called a *driver* or *I/O utility.* The driver must perform initialization functions and handle status and control, as well as physically transfer the actual data.

J

Jump instruction (or Branch instruction). An instruction that places a new value in the program counter, thus departing from the normal one-step incrementing. Jump instructions may be conditional; that is, the new value may be placed in the program counter only if a condition holds.

Jump table. A table consisting of the starting addresses of executable routines, used to transfer control to one of them.

L

Label. A name attached to an instruction or statement in a program that identifies the location in memory of the machine language code or assignment produced from that instruction or statement.

Latch. A device that retains its contents until new data is specifically entered into it.

Leading edge (of a binary pulse). The edge that marks the beginning of a pulse.

Least significant bit. The rightmost bit in a group of bits, that is, bit 0 of a byte or a 16-bit word.

Library program. A program that is part of a collection of programs and is written and documented according to a standard format.

LIFO (last-in, first-out) memory. A memory that is organized according to the order in which elements are entered and from which elements can be retrieved only in the order opposite from that in which they were entered. *See also* Stack.

Linearization. The mathematical approximation of a function by a straight line between two points at which its values are known.

Linked list. A list in which each item contains a pointer (or *link*) to the next item. Also called a *chain* or *chained list.*

List. An ordered set of items.

Logical device. The input or output device to which a program refers. The actual or physical device is determined by looking up the logical device in an I/O device table — a table containing actual I/O addresses (or starting addresses for I/O drivers) corresponding to the logical device numbers.

Logical shift. A shift operation that moves zeros in at the end as the original data is shifted.

Longitudinal parity. See Checksum.

Logical sum. A binary sum with no carries between bit positions. *See also* Checksum, EXCLUSIVE OR function.

Longitudinal redundancy check (LRC). *See* Checksum.

Lookup table. An array of data organized so that the answer to a problem may be determined merely by selecting the correct entry (without any calculations).

Low-level language. A computer language in which each statement is translated directly into a single machine language instruction.

M

Machine language. The programming language that the computer can execute directly with no translation other than numeric conversions.

Maintenance (of programs). Updating and correcting computer programs that are in use.

Majority logic. A combinational logic function that is true when more than half the inputs are true.

Manual mode (of a peripheral chip). An operating mode in which the chip produces control signals only when specifically directed to do so by a program.

Mark. The 1 state on a serial data communications line.

Mask. A bit pattern that isolates one or more bits from a group of bits.

Maskable interrupt. An interrupt that the system can disable.

Memory capacity. The total number of different memory addresses (usually specified in terms of bytes) that can be attached to a particular computer.

Microcomputer. A computer that has a microprocessor as its central processing unit.

Microprocessor. A complete central processing unit for a computer constructed from one or a few integrated circuits.

Mnemonic. A memory jogger, a name that suggests the actual meaning or purpose of the object to which it refers.

Modem (Modulator/demodulator). A device that adds or removes a carrier frequency, thereby allowing data to be transmitted on a high-frequency channel or received from such a channel.

Modular programming. A programming method whereby the overall program is divided into logically separate sections or *modules.*

Module. A part or section of a program.

Monitor. A program that allows the computer user to enter programs and data, run programs, examine the contents of the computer's memory and registers, and utilize the computer's peripherals. *See also* Operating system.

Most significant bit. The leftmost bit in a group of bits, that is, bit 7 of a byte or bit 15 of a 16-bit word.

Multifunction device. A device that performs more than one function in a computer system; the term commonly refers to devices containing memory, input/output ports, timers, etc., such as the 6530, 6531, and 6532 devices.

Multitasking. Used to execute many tasks during a single period of time, usually by working on each one for a specified part of the period and suspending tasks that must wait for input, output, the completion of other tasks, or external events.

Murphy's Law. The famous maxim that "whatever can go wrong, will."

N

Negate. Finds the two's complement (negative) of a number.

Negative edge (of a binary pulse). A 1-to-0 transition.

Negative flag. See Sign flag.

Negative logic. Circuitry in which a logic zero is the active or ON state.

Nesting. Constructing programs in a hierarchical manner with one level contained within another, and so forth. The nesting level is the number of transfers of control required to reach a particular part of a program without ever returning to a higher level.

Nibble (or nybble). A unit of four bits. A byte (eight bits) may be described as consisting of a high nibble (four most significant bits) and a low nibble (four least significant bits).

Nine's complement. The result of subtracting a decimal number from a number having nines in each digit position.

Nonmaskable interrupt. An interrupt that cannot be disabled within the CPU.

Nonvolatile memory. A memory that retains its contents when power is removed.

No-op (or no operation). An instruction that does nothing other than increment the program counter.

Normalization (of numbers). Adjusting a number into a regular or standard format. A typical example is the scaling of a binary fraction so that its most significant bit is 1.

O

Object code (or object program). The program that is the output of a translator program, such as an assembler. Usually it is a machine language program ready for execution.

Odd parity. A 1-bit error-detecting code that makes the total number of 1 bits in a unit of data (including the parity bit) odd.

Offset. Distance from a starting point or base address.

One's complement. A bit-by-bit logical complement of a number, obtained by replacing each 0 bit with a 1 and each 1 bit with a 0.

One-shot. A device that produces a pulse output of known duration in response to a pulse input. A timer operates in a *one-shot mode* when it indicates the end of a single interval of known duration.

Open (a file). Make a file ready for use. The user generally must open a file before working with it.

Operating system (OS). A computer program that controls the overall operations of a computer and performs such functions as assigning places in memory to

programs and data, scheduling the execution of programs, processing interrupts, and controlling the overall input/output system. Also known as a monitor, executive, or master-control program, although the term *monitor* is usually reserved for a simple operating system with limited functions.

Operation code (op code). The part of an instruction that specifies the operation to be performed.

OS. See Operating system.

Output register. In a PIA or VIA, the actual input/output port. Also called a *data register* or a *peripheral register.*

Overflow (of a stack). Exceeding the amount of memory allocated to a stack.

Overflow, two's complement. See Two's complement overflow.

P

P register. See Processor status register, Program counter. Most 6502 reference material abbreviates program counter as PC and processor status register as P, but some refer to the program counter as P and the processor status (flag) register as F.

Packed decimal. A binary-coded decimal format in which each 8-bit byte contains two decimal digits.

Page. A subdivision of the memory. In 6502 terminology, a page is a 256-byte section of memory in which all addresses have the same eight most significant bits (or page number). For example, page C6 consists of memory addresses C600 through C6FF.

Paged address. The identifier that characterizes a particular memory address on a known page. In 6502 terminology, this is the eight least significant bits of a memory address.

Page number. The identifier that characterizes a particular page of memory. In 6502 terminology, this is the eight most significant bits of a memory address.

Page 0. In 6502 terminology, the lowest 256 addresses in memory (addresses 0000 through 00FF).

Parallel interface. An interface between a CPU and input or output devices that handle data in parallel (more than one bit at a time).

Parameter. An item that must be provided to a subroutine or program in order for it to be executed.

Parity. A 1-bit error-detecting code that makes the total number of 1 bits in a unit of data, including the parity bit, odd (odd parity) or even (even parity). Also called *vertical parity* or *vertical redundancy check* (VRC).

Passing parameters. Making the required parameters available to a subroutine.

Peripheral Interface. One of the 6500 family versions of a parallel interface; examples are the 6520, 6522, 6530, and 6532 devices.

Peripheral ready. A signal that is active when a peripheral can accept more data.

Peripheral register. In a PIA or VIA, the actual input or output port. Also called a *data register* or an *output* register.

Physical device. An actual input or output device, as opposed to a logical device.

PIA. (Peripheral Interface Adapter). The common name for the 6520 or 6820 device which consists of two bidirectional 8-bit I/O ports, two status lines, and two bidirectional status or control lines. The 6821 is a similar device.

Pointer. A storage place that contains the address of a data item rather than the item itself. A pointer tells where the item is located.

Polling. Determining which I/O devices are ready by examining the status of one device at a time.

Polling interrupt system. An interrupt system in which a program determines the source of a particular interrupt by examining the status of potential sources one at a time.

Pop. Removes an operand from a stack.

Port. The basic addressable unit of the computer's input/output section.

Positive edge (of a binary pulse). A 0-to-1 transition.

Postdecrementing. Decrementing an address register after using it.

Postincrementing. Incrementing an address register after using it.

Postindexing. See Indirect indexed addressing.

Power fail interrupt. An interrupt that informs the CPU of an impending loss of power.

Predecrementing. Decrements an address register before using it.

Preincrementing. Increments an address register before using it.

Preindexing. See Indexed indirect addressing.

Priority interrupt system. An interrupt system in which some interrupts have precedence over others, that is, they will be serviced first or can interrupt the others' service routines.

Processor status (P or F) *register.* A register that defines the current state of a computer,. often containing various bits indicating internal conditions. Other names for this register include condition code register, flag (F) register, status register, and status word.

Program counter (PC or P register). A register that contains the address of the next instruction to be fetched from memory.

Programmable I/O device. An I/O device that can have its mode of operation determined by loading registers under program control.

Programmable peripheral chip. A chip that can operate in a variety of modes; its current operating mode is determined by loading control registers under program control.

Programmable timer. A device that can handle a variety of timing tasks, including the generation of delays, under program control.

Program relative addressing. A form of relative addressing in which the base address is the program counter. Use of this form of addressing makes it easy to move programs from one place in memory to another.

Programmed input/output. Input or output performed under program control without using interrupts or other special hardware techniques.

Protocol. See Data-link control.

Pseudo-operation (or pseudo-op or pseudo-instruction). An assembly language operation code that directs the assembler to perform some action but does not result in the generation of a machine language instruction.

Pull. Removes an operand from a stack, same as *pop.*

Push. Stores an operand in a stack.

Q

Queue. A set of tasks, storage addresses, or other items that are used in a first-in, first-out manner; that is, the first item entered in the queue is the first to be removed.

Queue header. A set of storage locations describing the current location and status of a queue.

R

RAM. See Random-access memory.

Random-access memory (RAM). A memory that can be both read and altered (written) in normal operation.

Read-only memory (ROM). A memory that can be read but not altered in normal operation.

Ready for data. A signal that is active when the receiver can accept more data.

Real-time. In synchronization with the actual occurrence of events.

Real-time clock. A device that interrupts a CPU at regular time intervals.

Real-time operating system. An operating system that can act as a supervisor for programs that have real-time requirements. May also be referred to as a *real-time executive* or *real-time monitor.*

Reentrant. A program or routine that can be executed concurrently while the same routine is being interrupted or otherwise held in abeyance.

Register. A storage location inside the CPU.

Relative addressing. An addressing mode in which the address specified in the instruction is the offset from a base address.

Relative offset. The difference between the actual address to be used in an instruction and the current value of the program counter.

Relocatable. Can be placed anywhere in memory without changes; that is, a program that can occupy any set of consecutive memory addresses.

Return (from a subroutine). Transfers control back to the program that originally called the subroutine and resumes its execution.

RIOT. (ROM/I/O/timer or RAM/I/O/timer). A device containing memory (ROM or RAM), I/O ports, and timers.

ROM. See Read-only memory.

Rotate. A shift operation that treats the data as if it were arranged in a circle, that is, as if the most significant and least significant bits were connected either directly or through a Carry bit.

Row major order. Storing elements of a multidimensional array in a linear memory by changing the indexes starting with the rightmost first. That is, if the elements are A(I,J,K) and begin with A(0,0,0), the order is A(0,0,0), A(0,0,1), ...,A(0,1,0), A(0,1,1),... The opposite technique (change leftmost index first) is called *column major order.*

RRIOT. ROM/RAM/I/O/timer, a device containing read-only memory, read/write memory, I/O ports, and timers.

RS-232 (or EIA RS-232). A standard interface for the transmission of serial digital data, sponsored by the Electronic Industries Association of Washington, D.C. It has been partially superseded by RS-449.

S

Scheduler. A program that determines when other programs should be started and terminated.

Scratchpad. An area of memory that is especially easy and quick to use for storing variable data or intermediate results. Page 0 is generally used as a scratchpad in 6502-based computers.

SDLC (Synchronous Data Link Control). The successor protocol to BSC for IBM computers and terminals.

Semaphore. See Flag.

Serial. One bit at a time.

Serial interface. An interface between a CPU and input or output devices that handle data serially. Serial interfaces commonly used in 6502-based computers are the 6551 and 6850 devices. *See also* UART.

Shift instruction. An instruction that moves all the bits of the data by a certain number of bit positions, just as in a shift register.

Signed number. A number in which one or more bits represent whether the number is positive or negative. A common format is for the most significant bit to represent the sign (0 = positive, 1 = negative).

Sign extension. The process of copying the sign (most significant) bit to the right as in an arithmetic shift. Sign extension preserves the sign when two's complement numbers are being divided or normalized.

Sign flag. A flag that contains the most significant bit of the result of the previous operation. It is sometimes called a *negative flag,* since a value of 1 indicates a negative signed number.

Sign function. A function that is 0 if its parameter is positive and 1 if its parameter is negative..

Software delay. A program that has no function other than to waste time.

Software interrupt. See Trap.

Software stack. A stack that is managed by means of specific instructions, as opposed to a hardware stack which the computer manages automatically.

Source code (or source program). A computer program written in assembly language or in a high-level language.

Space. The zero state on a serial data communications line.

Stack. A section of memory that can be accessed only in a last-in, first-out manner. That is, data can be added to or removed from the stack only through its top; new data is placed above the old data and the removal of a data item makes the item below it the new top.

Stack pointer. A register that contains the address of the top of a stack.The 6502's stack pointer contains the address on page 1 of the next available (empty) stack location.

Standard (or 8,4,2,1) *BCD.* A BCD representation in which the bit positions have the same weights as in ordinary binary numbers.

Standard teletypewriter. A teletypewriter that operates asynchronously at a rate of ten characters per second.

Start bit. A 1-bit signal that indicates the start of data transmission by an asynchronous device.

Static allocation (of memory). Assignment of fixed storage areas for data and programs, as opposed to *dynamic allocation* in which storage areas are assigned at the time when they are needed.

Status register. A register whose contents indicate the current state or operating mode of a device. *See also* Processor status register.

Status signal. A signal that describes the current state of a transfer or the operating mode of a device.

Stop bit. A 1-bit signal that indicates the end of data transmission by an asynchronous device.

String. An array (set of data) consisting of characters.

String functions. Procedures that allow the programmer to operate on data consisting of characters rather than numbers. Typical functions are insertion, deletion, concatenation, search, and replacement.

Strobe. A signal that identifies or describes another set of signals and that can be used to control a buffer, latch, or register.

Subroutine. A subprogram that can be executed (called) from more than one place in a main program.

Subroutine call. The process whereby a computer transfers control from its current program to a subroutine while retaining the information required to resume the current program.

Subroutine linkage. The mechanism whereby a computer retains the information required to resume its current program after it completes the execution of a subroutine.

Suspend (a task). Halts execution and preserves the status of the task until some future time.

Synchronization (or sync) character. A character that is used only to synchronize the transmitter and the receiver.

Synchronous. Operating according to an overall timing source or clock, that is, at regular intervals.

Systems software. Programs that perform administrative functions or aid in the development of other programs but do not actually perform any of the computer's ultimate workload.

T

Tail (of a queue). The location of the oldest item in the queue, that is, the earliest entry.

Task. A self-contained program that can serve as part of an overall system under the control of a supervisor.

Task status. The set of parameters that specify the current state of a task. A task can be suspended and resumed as long as its status is saved and restored.

Teletypewriter. A device containing a keyboard and a serial printer that is often used in communications and with computers. Also referred to as a Teletype (a registered trademark of Teletype Corporation of Skokie, Illinois) or TTY.

Ten's complement. The result of subtracting a decimal number from zero (ignoring the negative sign), the nine's complement plus one.

Terminator. A data item that has no function other than to signify the end of an array.

Threaded code. A program consisting of subroutines, each of which automatically transfers control to the next one upon its completion.

Timeout. A period during which no activity is allowed to proceed, an inactive period.

Top of the stack. The address containing the item most recently entered into the stack.

Trace. A debugging aid that provides information about a program while the program is being executed. The trace usually prints all or some of the intermediate results.

Trailing edge (of a binary pulse). The edge that masks the end of a pulse.

Translate instruction. An instruction that converts its operand into the corresponding entry in a table.

Transparent routine. A routine that operates without interfering with the operations of other routines.

Trap (or software interrupt). An instruction that forces a jump to a specific (CPU-dependent) address, often used to produce breakpoints or to indicate hardware or software errors.

True borrow. See Borrow.

Two's complement. A binary number that, when added to the original number in a binary adder, produces a zero result. The two's complement of a number may be obtained by subtracting the number from zero or by adding 1 to the one's complement.

Two's complement overflow. A situation in which a signed arithmetic operation produces a result that cannot be represented correctly — that is, the magnitude overflows into the sign bit.

U

UART (Universal Asynchronous Receiver/Transmitter). An LSI device that acts as an interface between systems that handle data in parallel and devices that handle data in asynchronous serial form.

Underflow (of a stack). Attempting to remove more data from a stack than has been entered into it.

Unsigned number. A number in which all the bits are used to represent magnitude.

Utility. A general-purpose program, usually supplied by the computer manufacturer or part of an operating system, that executes a standard or common operation such as sorting, converting data from one format to another, or copying a file.

V

Valid data. A signal that is active when new data is available to the receiver.

Vectored interrupt. An interrupt that produces an identification code (or *vector*) that the CPU can use to transfer control to the appropriate service routine. The process whereby control is transferred to the service routine is called *vectoring.*

Versatile Interface Adapter (VIA). The name commonly given to the 6522 parallel interface device; it consists of two 8-bit bidirectional I/O ports, four status and control lines, two 16-bit timers, and a shift register.

VIA. See Versatile Interface Adapter.

Volatile memory. A memory that loses its contents when power is removed.

W

Walking bit test. A procedure whereby a single 1 bit is moved through each bit position in an area of memory and a check is made as to whether it can be read back correctly.

Word. The basic grouping of bits that a computer can process at one time. In dealing with microprocessors, the term often refers to a 16-bit unit of data.

Word boundary. A boundary between 16-bit storage units containing two bytes of information. If information is being stored in word-length units, only pairs of bytes conforming to (aligned with) word boundaries contain valid information. Misaligned pairs of bytes contain one byte from one word and one byte from another.

Word-length. A length of 16 bits per item.

Wraparound. Organization in a circular manner as if the ends were connected. A storage area exhibits wraparound if operations on it act as if the boundary locations were contiguous.

Write-only register. A register that the CPU can change but cannot read. If a program must determine the contents of such a register, it must save a copy of the data placed there.

Z

Zero flag. A flag that is 1 if the last operation produced a result of zero and 0 if it did not.

Zero page. In 6502 terminology, the lowest 256 memory addresses (addresses 0000 through 00FF).

Zero page addressing. In 6502 terminology, a form of direct addressing in which the instruction contains only an 8-bit address on page 0. That is, zero is implied as the more significant byte of the direct address and need not be included specifically in the instruction.

Zero-page indexed addressing. A form of indexed addressing in which the instruction contains a base address on page 0. That is, zero is implied as the more significant byte of the base address and need not be included explicitly in the instruction.

Zoned decimal. A binary-coded decimal format in which each 8-bit byte contains only one decimal digit.

Index